ARM Assembly Language Programming

Peter Knaggs

April 7, 2016

Warning: Work in progress

As a work in progress this book *will* contain errors

Preface

Broadly speaking, you can divide the history of computers into four periods: the mainframe, the mini, the microprocessor, and the modern post-microprocessor. The *mainframe* era was characterized by computers that required large buildings and teams of technicians and operators to keep them going. More often than not, both academics and students had little direct contact with the mainframe—you handed a deck of punched cards to an operator and waited for the output to appear hours later. During the mainfame era, academics concentrated on languages and compilers, algorithms, and operating systems.

The *minicomputer* era put computers in the hands of students and academics, because university departments could now buy their own minis. As minicomputers were not as complex as mainframes and because students could get direct hands-on experience, many departments of computer science and electronic engineering taught students how to program in the native language of the computer—assembly language. In those days, the mid 1970s, assembly language programming was used to teach both the control of I/O devices, and the writing of programs (i.e., assembly language was taught rather like high level languages). The explosion of computer software had not taken place, and if you wanted software you had to write it yourself.

The late 1970s saw the introduction of the *microprocessor*. For the first time, each student was able to access a real computer. Unfortunately, microprocessors appeared before the introduction of low-cost memory (both primary and secondary). Students had to program microprocessors in assembly language because the only storage mechanism was often a ROM with just enough capacity to hold a simple single-pass assembler.

The advent of the low-cost microprocessor system (usually on a single board) ensured that virtually every student took a course on assembly language. Even today, most courses in computer science include a module on computer architecture and organization, and teaching students to write programs in assembly language forces them to understand the computer's architecture. However, some computer scientists who had been educated during the mainframe era were unhappy with the microprocessor, because they felt that the 8-bit microprocessor was a retrograde step—its architecture was far more primitive than the mainframes they had studied in the 1960s.

The 1990s is the *post-microprocessor* era. Today's personal computers have more power and storage capacity than many of yesterday's mainframes, and they have a range of powerful software tools that were undreamed of in the 1970s. Moreover, the computer science curriculum of the 1990s has exploded. In 1970 a student could be expected to be familiar with all field of computer science. Today, a student can be expected only to browse through the highlights.

The availability of high-performance hardware and the drive to include more and more new material in the curriculum, has put pressure on academics to justify what they teach. In particular, many are questioning the need for courses on assembly language.

If you regard computer science as being primarily concerned with the *use* of the computer, you can argue that assembly language is an irrelevance. Does the surgeon study metallurgy in order to understand how a scalpel operates? Does the pilot study thermodynamics to understand how a jet engine operates? Does the news reader study electronics to understand how the camera

operates? The answer to all these questions is "*no*". So why should we inflict assembly language and computer architecture on the student?

First, *education* is not the same as *training*. The student of computer science is not simply being trained to use a number of computer packages. A university course leading to a degree should also cover the *history* and the *theoretical basis* for the subject. Without a knowledge of computer architecture, the computer scientist cannot understand how computers have developed and what they are capable of.

Is assembly language today the same as assembly language yesterday?

Two factors have influenced the way in which we teach assembly language—one is the way in which microprocessors have changed, and the other is the use to which assembly language teaching is put. Over the years microprocessors have become more and more complex, with the result that the architecture and assembly language of a modern state-of-the-art microprocessor is radically different to that of an 8-bit machine of the late 1970s. When we first taught assembly language in the 1970s and early 1980s, we did it to demonstrate how computers operated and to give students hands-on experience of a computer. Since all students either have their own computer or have access to a computer lab, this role of the single-board computer is now obsolete. Moreover, assembly language programming once attempted to ape high-level language programming— students were taught algorithms such as sorting and searching in assembly language, as if assembly language were no more than the (desperately) poor person's C.

The argument for teaching assembly language programming today can be divided into two components: the underpinning of computer architecture and the underpinning of computer software.

Assembly language teaches how a computer works at the machine (i.e., register) level. It is therefore necessary to teach assembly language to all those who might later be involved in computer architecture—either by specifying computers for a particular application, or by designing new architectures. Moreover, the von Neumann machine's sequential nature teaches students the limitation of conventional architectures and, indirectly, leads them on to unconventional architectures (parallel processors, Harvard architectures, data flow computers, and even neural networks).

It is probably in the realm of software that you can most easily build a case for the teaching of assembly language. During a student's career, he or she will encounter a lot of *abstract* concepts in subjects ranging from programming languages, to operating systems, to real-time programming, to AI. The foundation of many of these concepts lies in assembly language programming and computer architecture. You might even say that assembly language provides *bottom-up* support for the *top-down* methodology we teach in high-level languages. Consider some of the following examples (taken from the teaching of Advanced RISC Machines Ltd (ARM) assembly language).

Data types
: Students come across data types in high-level languages and the effects of strong and weak data typing. Teaching an assembly language that can operate on bit, byte, word and long word operands helps students understand data types. Moreover, the ability to perform any type of assembly language operation on any type of data structure demonstrates the need for strong typing.

Addressing modes
: A vital component of assembly language teaching is addressing modes (literal, direct, and indirect). The student learns how pointers function and how pointers are manipulated. This aspect is particularly important if the student is to become a C programmer. Because an assembly language is unencumbered by data types, the students' view of pointers is much simplified by an assembly language. The ARM has complex addressing modes that support direct and indirect addressing, generated jump tables and handling of unknown memory offsets.

The stack and subroutines
How procedures are called, and parameters passed and returned from procedures. By using an assembly language you can readily teach the passing of parameters by *value* and by *reference*. The use of *local variables* and *re-entrant* programming can also be taught. This supports the teaching of task switching kernels in both operating systems and real-time programming.

Recursion
The recursive calling of subroutines often causes a student problems. You can use an assembly language, together with a suitable system with a tracing facility, to demonstrate how recursion operates. The student can actually observe how the stack grows as procedures are called.

Run-time support for high-level languages
A high-performance processor like the ARM provides facilities that support run-time checking in high-level languages. For example, the programming techniques document lists a series of programs that interface with 'C' and provide run-time checking for errors such as an attempt to divide a number by zero.

Protected-mode operation
Members of the ARM family operate in either a *priviledge mode* or a *user mode*. The operating system operates in the priviledge mode and all user (applications) programs run in the user mode. This mechanism can be used to construct *secure* or *protected* environments in which the effects of an error in one application can be prevented from harming the operating system (or other applications).

Input-output
Many high-level languages make it difficult to access I/O ports and devices directly. By using an assembly language we can teach students how to write device drivers and how to control interfaces. Most real interfaces are still programmed at the machine level by accessing registers within them.

All these topics can, of course, be taught in the appropriate courses (e.g., high-level languages, operating systems). However, by teaching them in an assembly language course, they pave the way for future studies, and also show the student exactly what is happening within the machine.

Conclusion

A strong case can be made for the continued teaching of assembly language within the computer science curriculum. However, an assembly language cannot be taught just as if it were another general-purpose programming language as it was once taught ten years ago. Perhaps more than any other component of the computer science curriculum, teaching an assembly language supports a wide range of topics at the heart of computer science. An assembly language should not be used just to illustrate algorithms, but to demonstrate what is actually happening inside the computer.

Acknowledgements

As usual there are many people without whom this book could not exist. The first of these is Stephen Welsh, without whom I may never have started this project and for reading and commenting on many revisions of the text. Andrew Main for finding me the time to work on it. Andrew Watson for his support, comments and ideas.

The students of the Computing and Software Engineering Management courses for unwittingly debugging the examples, exercises, and the development of the main text.

Contents

List of Programs

1 Introduction

A computer program is ultimately a series of numbers and therefore has very little meaning to a human being. In this chapter we will discuss the levels of human-like language in which a computer program may be expressed. We will also discuss the reasons for and uses of assembly language.

1.1 The Meaning of Instructions

The instruction set of a microprocessor is the set of binary inputs that produce defined actions during an instruction cycle. An instruction set is to a microprocessor what a function table is to a logic device such as a gate, adder, or shift register. Of course, the actions that the microprocessor performs in response to its instruction inputs are far more complex than the actions that logic devices perform in response to their inputs.

1.1.1 Binary Instructions

An instruction is a binary digit pattern — it must be available at the data inputs to the microprocessor at the proper time in order to be interpreted as an instruction. For example, when the ARM receives the binary pattern `111000000100` as the input during an instruction fetch operation, the pattern means subtract. Similary the microinstruction `111000001000` means add. Thus the 32 bit pattern `11100000010011101100000000001111` means:

"*Subtract R15 from R14 and put the answer in R12.*"

The microprocessor (like any other computer) only recognises binary patterns as instructions or data; it does not recognise characters or octal, decimal, or hexadecimal numbers.

1.2 A Computer Program

A program is a series of instructions that causes a computer to perform a particular task.

Actually, a computer program includes more than instructions, it also contains the data and the memory addresses that the microprocessor needs to accomplish the tasks defined by the instructions. Clearly, if the microprocessor is to perform an addition, it must have two numbers to add and a place to put the result. The computer program must determine the sources of the data and the destination of the result as well as the operation to be performed.

All microprocessors execute instructions sequentially unless an instruction changes the order of execution or halts the processor. That is, the processor gets its next instruction from the next higher memory address unless the current instruction specifically directs it to do otherwise.

Ultimately, every program is a set of binary numbers. For example, this is a snippet of an ARM program that adds the contents of memory locations 8094_{16} and 8098_{16} and places the result in memory location $809C_{16}$:

```
11100101100111110001000000010000
11100101100111110001000000001000
11100000100000010101000000000000
11100101100011110101000000001000
```

This is a machine language, or object, program. If this program were entered into the memory of an ARM-based microcomputer, the microcomputer would be able to execute it directly.

1.3 The Binary Programming Problem

There are many difficulties associated with creating programs as object, or binary machine language, programs. These are some of the problems:

- The programs are difficult to understand or debug. (Binary numbers all look the same, particularly after you have looked at them for a few hours.)
- The programs do not describe the task which you want the computer to perform in anything resembling a human-readable format.
- The programs are long and tiresome to write.
- The programmer often makes careless errors that are very difficult to locate and correct.

For example, the following version of the addition object program contains a single bit error. Try to find it:

```
11100101100111110001000000010000
11100101100111110001000000001000
11100000100000010101000000000000
11100110100011110101000000001000
```

Although the computer handles binary numbers with ease, people do not. People find binary programs long, tiresome, confusing, and meaningless. Eventually, a programmer may start remembering some of the binary codes, but such effort should be spent more productively.

1.4 Using Octal or Hexadecimal

We can improve the situation somewhat by writing instructions using octal or hexadecimal numbers, rather than binary. We will use hexadecimal numbers because they are shorter, and because they are the standard for the microprocessor industry. Table 1.1 defines the hexadecimal digits and their binary equivalents. The ARM program to add two numbers now becomes:

```
E59F1010
E59f0008
E0815000
E58F5008
```

At the very least, the hexadecimal version is shorter to write and not quite so tiring to examine.

Errors are somewhat easier to find in a sequence of hexadecimal digits. The erroneous version of the addition program, in hexadecimal form, becomes:

Hexadecimal Digit	Binary Equivalent	Decimal Equivalent
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

Table 1.1: Hexadecimal Conversion Table

```
E59F1010
E59f0008
E0815000
E68F5008
```

The mistake is far more obvious.

The hexadecimal version of the program is still difficult to read or understand; for example, it does not distinguish operations from data or addresses, nor does the program listing provide any suggestion as to what the program does. What does 3038 or 31C0 mean? Memorising a card full of codes is hardly an appetising proposition. Furthermore, the codes will be entirely different for a different microprocessor and the program will require a large amount of documentation.

1.5 Instruction Code Mnemonics

An obvious programming improvement is to assign a name to each instruction code. The instruction code name is called a "*mnemonic*" or memory jogger.

In fact, all microprocessor manufacturers provide a set of mnemonics for the microprocessor instruction set (they cannot remember hexadecimal codes either). You do not have to abide by the manufacturer's mnemonics; there is nothing sacred about them. However, they are standard for a given microprocessor, and therefore understood by all users. These are the instruction codes that you will find in manuals, cards, books, articles, and programs. The problem with selecting instruction mnemonics is that not all instructions have "obvious" names. Some instructions do (for example, `ADD`, `AND`, `ORR`), others have obvious contractions (such as `SUB` for subtraction, `EOR` for exclusive-OR), while still others have neither. The result is such mnemonics as `BIC`, `STMIA`, and even `MRS`. Most manufacturers come up with some reasonable names and some hopeless ones. However, users who devise their own mnemonics rarely do much better.

Along with the instruction mnemonics, the manufacturer will usually assign names to the CPU registers. As with the instruction names, some register names are obvious (such as `A` for Accumulator) while others may have only historical significance. Again, we will use the manufacturer's suggestions simply to promote standardisation.

If we use standard ARM instruction and register mnemonics, as defined by Advanced RISC Machines, our ARM addition program becomes:

```
LDR   R1, num1
LDR   R0, num2
ADD   R5, R1, R0
STR   R5, num3
```

The program is still far from obvious, but at least some parts are comprehensible. `ADD` is a considerable improvement over `E59F`. The `LDR` mnemonic does suggest loading data into a register or memory location. We now see that some parts of the program are operations and others are addresses. Such a program is an assembly language program.

1.6 The Assembler Program

How do we get the assembly language program into the computer? We have to translate it, either into hexadecimal or into binary numbers. You can translate an assembly language program by hand, instruction by instruction. This is called hand assembly.

The following table illustrates the hand assembly of the addition program:

Instruction Mnemonic	Register/Memory Location	Hexadecimal Equivalent
`LDR`	`R1, num1`	`E59F1010`
`LDR`	`R0, num2`	`E59F0008`
`ADD`	`R5, R1, R0`	`E0815000`
`STR`	`R5, num3`	`E58F5008`

Hand assembly is a rote task which is uninteresting, repetitive, and subject to numerous minor errors. Picking the wrong line, transposing digits, omitting instructions, and misreading the codes are only a few of the mistakes that you may make. Most microprocessors complicate the task even further by having instructions with different lengths. Some instructions are one word long while others may be two or three. Some instructions require data in the second and third words; others require memory addresses, register numbers, or who knows what?

Assembly is a rote task that we can assign to the microcomputer. The microcomputer never makes any mistakes when translating codes; it always knows how many words and what format each instruction requires. The program that does this job is an "*assembler*." The assembler program translates a user program, or "source" program written with mnemonics, into a machine language program, or "object" program, which the microcomputer can execute. The assembler's input is a source program and its output is an object program.

Assemblers have their own rules that you must learn. These include the use of certain markers (such as spaces, commas, semicolons, or colons) in appropriate places, correct spelling, the proper control of information, and perhaps even the correct placement of names and numbers. These rules are usually simple and can be learned quickly.

1.6.1 Additional Features of Assemblers

Early assemblers did little more than translate the mnemonic names of instructions and registers into their binary equivalents. However, most assemblers now provide such additional features as:

- Allowing the user to assign names to memory locations, input and output devices, and even sequences of instructions
- Converting data or addresses from various number systems (for example, decimal or hexadecimal) to binary and converting characters into their ASCII or EBCDIC binary codes
- Performing some arithmetic as part of the assembly process

- Telling the loader program where in memory parts of the program or data should be placed
- Allowing the user to assign areas of memory as temporary data storage and to place fixed data in areas of program memory
- Providing the information required to include standard programs from program libraries, or programs written at some other time, in the current program
- Allowing the user to control the format of the program listing and the input and output devices employed

1.6.2 Choosing an Assembler

All of these features, of course, involve additional cost and memory. Microcomputers generally have much simpler assemblers than do larger computers, but the tendency is always for the size of assemblers to increase. You will often have a choice of assemblers. The important criterion is not how many off-beat features the assembler has, but rather how convenient it is to use in normal practice.

1.7 Disadvantages of Assembly Language

The assembler does not solve all the problems of programming. One problem is the tremendous gap between the microcomputer instruction set and the tasks which the microcomputer is to perform. Computer instructions tend to do things like add the contents of two registers, shift the contents of the Accumulator one bit, or place a new value in the Program Counter. On the other hand, a user generally wants a microcomputer to do something like print a number, look for and react to a particular command from a teletypewriter, or activate a relay at the proper time. An assembly language programmer must translate such tasks into a sequence of simple computer instructions. The translation can be a difficult, time-consuming job.

Furthermore, if you are programming in assembly language, you must have detailed knowledge of the particular microcomputer that you are using. You must know what registers and instructions the microcomputers has, precisely how the instructions affect the various registers, what addressing methods the computer uses, and a mass of other information. None of this information is relevant to the task which the microcomputer must ultimately perform.

In addition, assembly language programs are not portable. Each microcomputer has its own assembly language which reflects its own architecture. An assembly language program written for the ARM will not run on a 486, Pentium, or Z8000 microprocessor. For example, the addition program written for the Z8000 would be:

```
LD   R0,%6000
ADD  R0,%6002
LD   %6004,R0
```

The lack of portability not only means that you will not be able to use your assembly language program on a different microcomputer, but also that you will not be able to use any programs that were not specifically written for the microcomputer you are using. This is a particular drawback for new microcomputers, since few assembly language programs exist for them. The result, too frequently, is that you are on your own. If you need a program to perform a particular task, you are not likely to find it in the small program libraries that most manufacturers provide. Nor are you likely to find it in an archive, journal article, or someone's old program File. You will probably have to write it yourself.

1.8 High-Level Languages

The solution to many of the difficulties associated with assembly language programs is to use, insted, *high-level* or *procedure-oriented* langauges. Such languages allow you to describe tasks in forms that are problem-oriented rather than computer-oriented. Each statement in a high-level language performs a recognisable function; it will generally correspond to many assembly language instruction. A program called a compiler translates the high-level language source program into object code or machine language instructions.

Many different hgih-level languages exist for different types of tasks. If, for exampe, you can express what you want the computer to do in algebraic notation, you can write your FORTRAN (*For*mula *Trans*lation Language), the oldest of the high-level languages. Now, if you want to add two numbers, you just tell the computer:

```
sum = num1 + num2;
```

That is a lot simpler (and shorter) than either the equivalent machine language program or the equivalent assembly language program. Other high-level languages include COBOL (for business applications), BASIC (a cut down version of FORTRAN designed to prototype ideas before codeing them in full), C (a systems-programming language), C++ and JAVA (object-orientated general development languages).

1.8.1 Advantages of High-Level Languages

Clearly, high-level languages make program easier and faster to write. A common estimate is that a programmer can write a program about ten times as fast in a high-level langauge as in assembly language. That is just writing the program; it does not include problem definition, program design, debugging testing or documentation, all of which become simpler and faster. The high-level language program is, for instance, partly self-documenting. Even if you do not know FORTRAN, you could probably tell what the statement illustrated above does.

Machine Independence

High-level languages solve many other problems associated with assembly language programming. The high-level language has its own syntax (usually defined by an international standard). The language does not mention the instruction set, registers, or other features of a particular computer. The compiler takes care of all such details. Programmers can concentrate on their own tasks; they do not need a detailed understanding of the underlying CPU architecture — for that matter, they do not need to know anything about the computer the are programming.

Portability

Programs written in a high-level language are portable — at least, in theory. They will run on any computer that has a standard compiler for that language.

At the same time, all previous programs written in a high-level language for prior computers and available to you when programming a new computer. This can mean thousands of programs in the case of a common language like C.

1.8.2 Disadvantages of High-Level Languages

If all the good things we have said about high-level languages are true — if you can write programs faster and make them portable besides — why bother with assebly languages? Who wants to worry about registers, instruction codes, mnemonics, and all that garbage! As usual, there are disadvantages that balance the advantages.

Syntax

One obvious problem is that, as with assembly language, you have to learn the "rules" or *syntax* of any high-level language you want to use. A high-level langauge has a fairly complicated set of rules. You will find that it takes a lot of time just to get a program that is syntactically correct (and even then it probably will not do what you want). A high-level computer language is like a foreign language. If you have talent, you will get used to the rules and be able to turn out programs that the compiler will accept. Still, learning the rules and trying to get the program accepted by the compiler does not contribute directly to doing your job.

Cost of Compilers

Another obvious problem is that you need a compiler to translate program written in a high-level language into machine language. Compilers are expensive and use a large amount of memory. While most assemblers occupy only a few KBytes of memory, compilers would occupy far larger amounts of memory. A compiler could easily require over four times as much memory as an assembler. So the amount of overhead involved in using the compiler is rather large.

Adapting Tasks to a Language

Furthermore, only some compilers will make the implementation of your task simpler. Each language has its own target proglem area, for example, FORTRAN is well-suited to problems that can be expressed as algebraic formulas. If however, your problem is controlling a display terminal, editing a string of characters, or monitoring an alarm system, your problem cannot be easily expressed. In fact, formulating the solution in FORTRAN may be more awkward and more difficult than formulating it in assembly language. The answer is, of course, to use a more suitable high-level language. Languages specifically designed for tasks such as those mentioned above do exist — they are called system implementation languages. However, these languages are less widely used.

Inefficiency

High-level languages do not produce very efficient machine language program. The basic reason for this is that compilation is an automatic process which is riddled with compromises to allow for many ranges of possibilities. The compiler works much like a computerised language translator — sometimes the words are right but the sentence structures are awkward. A simpler compiler connot know when a variable is no longer being used and can be discarded, when a register should be used rather than a memory location, or when variables have simple relationships. The experienced programmer can take advantage of shortcuts to shorten execution time or reduce memory usage. A few compiler (known as optimizing cmpilers) can also do this, but such compilers are much larger than regular compilers.

1.9 Which Level Should You Use?

Which language level you use depends on your particulr application. Let us briefly note some of the factors which may favor particular levels:

1.9.1 Applications for Machine Language

Virtually no one programs in machine language because it wastes human time and is difficult to document. An assembler costs very little and greatly reduces programming time.

1.9.2 Applications for Assembly Language

- Limited data processing
- High-volume applications
- Real-Time control applications
- Short to moderate-sized programs
- Application where memory cost is a factor
- Applications involving more input/output or control than computation

1.9.3 Applications for High-Level Language

- Long programs
- Low-volume applications
- Programs which are expected to undergo many changes
- Compatibility with similar applications using larger computers
- Applications involing more computation than input/output or control
- Applications where the amout of memory required is already very large
- Availability of a specific program in a high-level language which can be used in the application.

1.9.4 Other Considerations

Many other factors are also important, such as the availability of a large computer for use in development, experience with particular languages, and compatibility with other applications.

If hardware will ultimately be the largest cost in your application, or if speed is critical, you should favor assembly language. But be prepared to spend much extra time in software development in exchange for lower memory costs and higher execution speeds. If software will be the largest cost in your application, you should favor a high-level language. But be prepared to spend the extra money required for the supporting hardware and software.

Of course, no one except some theorists will object if you use both assembly and high-level languages. You can write the program originally in a high-level language and then patch some sections in assembly language. However, most users prefer not to do this because it can create havoc in debugging, testing, and documentation.

1.10 Why Learn Assembler?

Given the advance of high-level languages, why do you need to learn assembly language programming? The reasons are:

1. Most industrial microcomputer users program in assembly language.
2. Many microcomputer users will continue to program in assembly language since they need the detailed control that it provides.
3. No suitable high-level language has yet become widely available or standardised.
4. Many application require the efficiency of assembly language.
5. An understanding of assembly language can help in evaluating high-level languages.
6. Almost all microcomputer programmers ultimately find that they need some knowledge of assembly language, most often to debug programs, write I/O routines, speed up or shorten critical sections of programs written in high-level languages, utilize or modify operating system functions, and undertand other people's programs.

The rest of these notes will deal exclusively with assembler and assembly language programming.

2 Assemblers

This chapter discusses the functions performed by assemblers, beginning with features common to most assemblers and proceeding through more elaborate capabilities such as macros and conditional assembly. You may wish to skim this chapter for the present and return to it when you feel more comfortable with the material.

As we mentioned, today's assemblers do much more than translate assembly language mnemonics into binary codes. But we will describe how an assembler handles the translation of mnemonics before describing additional assembler features. Finally we will explain how assemblers are used.

2.1 Fields

Assembly language instructions (or "statements") are divided into a number of "*fields*".

The operation code field is the only field which can never he empty; it always contains either an instruction mnemonic or a directive to the assembler, sometimes called a "pseudo-instruction," "pseudo-operation," or "pseudo-op."

The operand or address field may contain an address or data, or it may be blank.

The comment and label fields are optional. A programmer will assign a label to a statement or add a comment as a personal convenience: namely, to make the program easier to read and use.

Of course, the assembler must have some way of telling where one field ends and another begins. Assemblers often require that each field start in a specific column. This is a "fixed format." However, fixed formats are inconvenient when the input medium is paper tape; fixed formats are also a nuisance to programmers. The alternative is a "free format" where the fields may appear anywhere on the line.

2.1.1 Delimiters

If the assembler cannot use the position on the line to tell the fields apart, it must use something else. Most assemblers use a special symbol or "delimiter" at the beginning or end of each field.

Label Field	Operation Code or Mnemonic Field	Operand or Address Field	Comment Field
VALUE1	DCW	0x201E	;FIRST VALUE
VALUE2	DCW	0x0774	;SECOND VALUE
RESULT	DCW	1	;16-BIT STORAGE FOR ADDITION RESULT
START	MOV	R0, VALUE1	;GET FIRST VALUE
	ADD	R0, R0, VALUE2	;ADD SECOND VALUE TO FIRST VALUE
	STR	RESULT, R0	;STORE RESULT OF ADDITION
NEXT:	?	?	;NEXT INSTRUCTION

label ⟨*whitespace*⟩ instruction ⟨*whitespace*⟩ ; comment

whitespace	Between label and operation code, between operation code and address, and before an entry in the comment field
comma	Between operands in the address field
asterisk	Before an entire line of comment
semicolon	Marks the start of a comment on a line that contains preceding code

Table 2.1: Standard ARM Assembler Delimiters

The most common delimiter is the space character. Commas, periods, semicolons, colons, slashes, question marks, and other characters that would not otherwise be used in assembly language programs also may serve as delimiters. The general form of layout for the ARM assembler is:

You will have to exercise a little care with delimiters. Some assemblers are fussy about extra spaces or the appearance of delimiters in comments or labels. A well-written assembler will handle these minor problems, but many assemblers are not well-written. Our recommendation is simple: avoid potential problems if you can. The following rules will help:

- Do not use extra spaces, in particular, do not put spaces after commas that separate operands, even though the ARM assembler allows you to do this.
- Do not use delimiter characters in names or labels.
- Include standard delimiters even if your assembler does not require them. Then it will be more likely that your programs are in correct form for another assembler.

2.1.2 Labels

The label field is the first field in an assembly language instruction; it may be blank. If a label is present, the assembler defines the label as equivalent to the address into which the first byte of the object code generated for that instruction will be loaded. You may subsequently use the label as an address or as data in another instruction's address field. The assembler will replace the label with the assigned value when creating an object program.

The ARM assembler requires labels to start at the first character of a line. However, some other assemblers also allow you to have the label start anywhere along a line, in which case you must use a colon (`:`) as the delimiter to terminate the label field. Colon delimiters are not used by the ARM assembler.

Labels are most frequently used in Branch or SWI instructions. These instructions place a new value in the program counter and so alter the normal sequential execution of instructions. `B` 150_{16} means "*place the value* 150_{16} *in the program counter*." The next instruction to be executed will be the one in memory location 150_{16}. The instruction `B START` means "*place the value assigned to the label* `START` *in the program counter*." The next instruction to be executed will be the on at the address corresponding to the label `START`. Figure 2.1 contains an example.

Why use a label? Here are some reasons:

- A label makes a program location easier to find and remember.
- The label can easily be moved, if required, to change or correct a program. The assembler will automatically change all instructions that use the label when the program is reassembled.
- The assembler can relocate the whole program by adding a constant (a "relocation constant") to each address in which a label was used. Thus we can move the program to allow for the insertion of other programs or simply to rearrange memory.

Assembly language Program

```
START   MOV   R0, VALUE1
          .
          .   (Main Program)
          .
        BAL   START
```

When the machine language version of this program is executed, the instruction `B START` causes the address of the instruction labeled `START` to be placed in the program counter That instruction will then be executed.

Figure 2.1: Assigning and Using a Label

- The program is easier to use as a library program; that is, it is easier for someone else to take your program and add it to some totally different program.
- You do not have to figure out memory addresses. Figuring out memory addresses is particularly difficult with microprocessors which have instructions that vary in length.

You should assign a label to any instruction that you might want to refer to later.

The next question is how to choose a label. The assembler often places some restrictions on the number of characters (usually 5 or 6), the leading character (often must be a letter), and the trailing characters (often must be letters, numbers, or one of a few special characters). Beyond these restrictions, the choice is up to you.

Our own preference is to use labels that suggest their purpose, i.e., mnemonic labels. Typical examples are `ADDW` in a routine that adds one word into a sum, `SRCHETX` in a routine that searches for the ASCII character ETX, or `NKEYS` for a location in data memory that contains the number of key entries. Meaningful labels are easier to remember and contribute to program documentation. Some programmers use a standard format for labels, such as starting with `L0000`. These labels are self-sequencing (you can skip a few numbers to permit insertions), but they do not help document the program.

Some label selection rules will keep you out of trouble. We recommend the following:

- Do not use labels that are the same as operation codes or other mnemonics. Most assemblers will not allow this usage; others will, but it is confusing.
- Do not use labels that are longer than the assembler recognises. Assemblers have various rules, and often ignore some of the characters at the end of a long label.
- Avoid special characters (non-alphabetic and non-numeric) and lower-case letters. Some assemblers will not permit them; others allow only certain ones. The simplest practice is to stick to capital letters and numbers.
- Start each label with a letter. Such labels are always acceptable.
- Do not use labels that could be confused with each other. Avoid the letters `I`, `O`, and `Z` and the numbers `0`, `1`, and `2`. Also avoid things like `XXXX` and `XXXXX`. Assembly programming is difficult enough without tempting fate or Murphy's Law.
- When you are not sure if a label is legal, do not use it. You will not get any real benefit from discovering exactly what the assembler will accept.

These are recommendations, not rules. You do not have to follow them but don't blame us if you waste time on unnecessary problems.

2.2 Operation Codes (Mnemonics)

One main task of the assembler is the translation of mnemonic operation codes into their binary equivalents. The assembler performs this task using a fixed table much as you would if you were doing the assembly by hand.

The assembler must, however, do more than just translate the operation codes. It must also somehow determine how many operands the instruction requires and what type they are. This may be rather complex — some instructions (like a Stop) have no operands, others (like a Jump instruction) have one, while still others (like a transfer between registers or a multiple-bit shift) require two. Some instructions may even allow alternatives; for example, some computers have instructions (like Shift or Clear) which can either apply to a register in the CPU or to a memory location. We will not discuss how the assembler makes these distinctions; we will just note that it must do so.

2.3 Directives

Some assembly language instructions are not directly translated into machine language instructions. These instructions are directives to the assembler; they assign the program to certain areas in memory, define symbols, designate areas of memory for data storage, place tables or other fixed data in memory, allow references to other programs, and perform minor housekeeping functions.

To use these assembler directives or pseudo-operations a programmer places the directive's mnemonic in the operation code field, and, if the specified directive requires it, an address or data in the address field.

The most common directives are:

DEFINE CONSTANT (Data)
EQUATE (Define)
AREA
DEFINE STORAGE (Reserve)

Different assemblers use different names for those operations but their functions are the same. Housekeeping directives include:

END LIST FORMAT TTL PAGE INCLUDE

We will discuss these pseudo-operations briefly, although their functions are usually obvious.

2.3.1 The DEFINE CONSTANT (Data) Directive

The DEFINE CONSTANT directive allows the programmer to enter fixed data into program memory. This data may include:

- Names
- Messages
- Commands
- Tax tables
- Thresholds
- Test patterns
- Lookup tables
- Standard forms
- Masking patterns
- Weighting factors
- Conversion factors
- Key identifications
- Subroutine addresses
- Code conversion tables
- Identification patterns
- State transition tables
- Synchronisation patterns
- Coefficients for equations
- Character generation patterns
- Characteristic times or frequencies

The define constant directive treats the data as a permanent part of the program.

The format of a define constant directive is usually quite simple. An instruction like:

```
DZCON    DCW     12
```

will place the number 12 in the next available memory location and assign that location the name DZCON. Every DC directive usually has a label, unless it is one of a series. The data and label may take any form that the assembler permits.

More elaborate define constant directives that handle a large amount of data at one time are provided, for example:

```
EMESS    DCB     'ERROR'
SQRS     DCW     1,4,9,16,25
```

A single directive may fill many bytes of program memory, limited perhaps by the length of a line or by the restrictions of a particular assembler. Of course, you can always overcome any restrictions by following one define constant directive with another:

```
MESSG    DCB      "NOW IS THE "
         DCB      "TIME FOR ALL "
         DCB      "GOOD MEN "
         DCB      "TO COME TO THE "
         DCB      "AID OF THEIR "
         DCB      "COUNTRY", 0 ;note the '0' terminating the string
```

Microprocessor assemblers typically have some variations of standard define constant directives. Define Byte or DCB handles 8-bit numbers; Define Word or DCW handles 32-bit numbers or addresses. Other special directives may handle character-coded data. The ARM assembler also defines DCD to (Define Constant Data) which may be used in place of DCW.

2.3.2 The EQUATE Directive

The EQUATE directive allows the programmer to equate names with addresses or data. This pseudo-operation is almost always given the mnemonic EQU. The names may refer to device addresses, numeric data, starting addresses, fixed addresses, etc.

The EQUATE directive assigns the numeric value in its operand field to the label in its label field. Here are two examples:

```
TTY      EQU     5
LAST     EQU     5000
```

Most assemblers will allow you to define one label in terms of another, for example:

```
LAST     EQU     FINAL
ST1      EQU     START+1
```

The label in the operand field must, of course, have been previously defined. Often, the operand field may contain more complex expressions, as we shall see later. Double name assignments (two names for the same data or address) may be useful in patching together programs that use different names for the same variable (or different spellings of what was supposed to be the same name).

Note that an EQU directive does not cause the assembler to place anything in memory. The assembler simply enters an additional name into a table (called a "symbol table") which the assembler maintains.

When do you use a name? The answer is: whenever you have a parameter that you might want to change or that has some meaning besides its ordinary numeric value. We typically assign names to

time constants, device addresses, masking patterns, conversion factors, and the like. A name like `DELAY`, `TTY`, `KBD`, `KROW`, or `OPEN` not only makes the parameter easier to change, but it also adds to program documentation. We also assign names to memory locations that have special purposes; they may hold data, mark the start of the program, or be available for intermediate storage.

What name do you use? The best rules are much the same as in the case of labels, except that here meaningful names really count. Why not call the teletypewriter `TTY` instead of `X15`, a bit time delay `BTIME` or `BTDLY` rather than `WW`, the number of the "GO" key on a keyboard `GOKEY` rather than `HORSE`? This advice seems straightforward, but a surprising number of programmers do not follow it.

Where do you place the EQUATE directives? The best place is at the start of the program, under appropriate comment headings such as I/O ADDRESSES, TEMPORARY STORAGE, TIME CONSTANTS, or PROGRAM LOCATIONS. This makes the definitions easy to find if you want to change them. Furthermore, another user will be able to look up all the definitions in one centralised place. Clearly this practice improves documentation and makes the program easier to use.

Definitions used only in a specific subroutine should appear at the start of the subroutine.

2.3.3 The AREA Directive

The AREA directive allows the programmer to specify the memory locations where programs, subroutines, or data will reside. Programs and data may be located in different areas of memory depending on the memory configuration. Startup routines interrupt service routines, and other required programs may be scattered around memory at fixed or convenient addresses.

The assembler maintains a location counter (comparable to the computer's program counter) which contains the location in memory of the instruction or data item being processed. An area directive causes the assembler to place a new value in the location counter, much as a Jump instruction causes the CPU to place a new value in the program counter. The output from the assembler must not only contain instructions and data, but must also indicate to the loader program where in memory it should place the instructions and data.

Microprocessor programs often contain several `AREA` statements for the following purposes:

- Reset (startup) address
- Interrupt service addresses
- Trap (software interrupt) addresses
- RAM storage
- Stack
- Main program
- Subroutines
- Input/Output

Still other origin statements may allow room for later insertions, place tables or data in memory, or assign vacant memory space for data buffers. Program and data memory in microcomputers may occupy widely separate addresses to simplify the hardware. Typical origin statements are:

```
AREA     RESET
AREA    $1000
AREA     INT3
```

The assembler will assume a fake address if the programmer does not put in an `AREA` statement. The AREA statement at the start of an ARM program is required, and its absence will cause the assembly to fail.

2.3.4 Housekeeping Directives

There are various assembler directives that affect the operation of the assembler and its program listing rather than the object program itself. Common directives include:

`END`, marks the end of the assembly language source program. This must appear in the file or a "missing END directive" error will occur.

`INCLUDE` will include the contents of a named file into the current file. When the included file has been processed the assembler will continue with the next line in the original file. For example the following line

```
INCLUDE  MATH.S
```

will include the content of the file math.s at that point of the file.

You should never use a lable with an include directive. Any labels defined in the included file will be defined in the current file, hence an error will be reported if the same label appears in both the source and include file.

An include file may itself include other files, which in turn could include other files, and so on, however, the level of includes the assembler will accept is limited. It is not recommended you go beyond three levels for even the most complex of software.

2.3.5 When to Use Labels

Users often wonder if or when they can assign a label to an assembler directive. These are our recommendations:

1. All `EQU` directives must have labels; they are useless otherwise, since the purpose of an `EQU` is to define its label.
2. Define Constant and Define Storage directives usually have labels. The label identifies the first memory location used or assigned.
3. Other directives should not have labels.

2.4 Operands and Addresses

The assembler allow the programmer a lot of freedom in describing the contents of the operand or address field. But remember that the assembler has built-in names for registers and instructions and may have other built-in names. We will now describe some common options for the operand field.

2.4.1 Decimal Numbers

The assembler assume all numbers to be decimal unless they are marked otherwise. So:

```
ADD     100
```

means "add the contents of memory location 100_{10} to the contents of the Accumulator."

2.4.2 Other Number Systems

The assembler will also accept hexadecimal entries. But you must identify these number systems in some way: for example, by preceding the number with an identifying character.

2_*nnn*	Binary	Base 2
8_*nnn*	Octal	Base 8
nnn	Decimal	Base 10
0x*nnn*	Hexadecimal	Base 16

It is good practice to enter numbers in the base in which their meaning is the clearest: that is, decimal constants in decimal; addresses and BCD numbers in hexadecimal; masking patterns or bit outputs in hexadecimal.

2.4.3 Names

Names can appear in the operand field; they will be treated as the data that they represent. Remember, however, that there is a difference between operands and addresses. In an ARM assembly language program the sequence:

```
FIVE    EQU     5
        ADD     R2, #FIVE
```

will add the contents of memory location FIVE (not necessarily the number 5) to the contents of data register `R2`.

2.4.4 Character Codes

The assembler allows text to be entered as ASCII strings. Such strings must be surrounded with double quotation marks, unless a single ASCII character is quoted, when single qoutes may be used exactly as in 'C'. We recommend that you use character strings for all text. It improves the clarity and readability of the program.

2.4.5 Arithmetic and Logical Expressions

Assemblers permit combinations of the data forms described above, connected by arithmetic, logical, or special operators. These combinations are called expressions. Almost all assemblers allow simple arithmetic expressions such as `START+1`. Some assemblers also permit multiplication, division, logical functions, shifts, etc. Note that the assembler evaluates expressions at assembly time; if a symbol appears in an expression, the address is used (i.e., the location counter or EQUATE value).

Assemblers vary in what expressions they accept and how they interpret them. Complex expressions make a program difficult to read and understand.

2.4.6 General Recommendations

We have made some recommendations during this section but will repeat them and add others here. In general, the user should strive for clarity and simplicity. There is no payoff for being an expert in the intricacies of an assembler or in having the most complex expression on the block. We suggest the following approach:

- Use the clearest number system or character code for data.
- Masks and BCD numbers in decimal, ASCII characters in octal, or ordinary numerical constants in hexadecimal serve no purpose and therefore should not be used.
- Remember to distinguish data from addresses.

- Don't use offsets from the location counter.
- Keep expressions simple and obvious. Don't rely on obscure features of the assembler.

2.5 Comments

All assemblers allow you to place comments in a source program. Comments have no effect on the object code, but they help you to read, understand, and document the program. Good commenting is an essential part of writing computer programs, programs without comments are very difficult to understand.

We will discuss commenting along with documentation in a later chapter, but here are some guidelines:

- Use comments to tell what application task the program is performing, not how the microcomputer executes the instructions.
- Comments should say things like "is temperature above limit?", "linefeed to TTY," or "examine load switch."
- Comments should not say things like "add 1 to Accumulator," "jump to Start," or "look at carry." You should describe how the program is affecting the system; internal effects on the CPU should be obvious from the code.
- Keep comments brief and to the point. Details should be available elsewhere in the documentation.
- Comment all key points.
- Do not comment standard instructions or sequences that change counters or pointers; pay special attention to instructions that may not have an obvious meaning.
- Do not use obscure abbreviations.
- Make the comments neat and readable.
- Comment all definitions, describing their purposes. Also mark all tables and data storage areas.
- Comment sections of the program as well as individual instructions.
- Be consistent in your terminology. You can (should) be repetitive, you need not consult a thesaurus.
- Leave yourself notes at points that you find confusing: for example, "remember carry was set by last instruction." If such points get cleared up later in program development, you may drop these comments in the final documentation.

A well-commented program is easy to use. You will recover the time spent in commenting many times over. We will try to show good commenting style in the programming examples, although we often over-comment for instructional purposes.

2.6 Types of Assemblers

Although all assemblers perform the same tasks, their implementations vary greatly. We will not try to describe all the existing types of assemblers, we will merely define the terms and indicate some of the choices.

A *cross-assembler* is an assembler that runs on a computer other than the one for which it assembles object programs. The computer on which the cross-assembler runs is typically a large computer with extensive software support and fast peripherals. The computer for which the cross-assembler assembles programs is typically a micro like the 6809 or MC68000.

When a new microcomputer is introduced, a cross-assembler is often provided to run on existing development systems. For example, ARM provide the 'Armulator' cross-assembler that will run on a PC development system.

A *self-assembler* or *resident assembler* is an assembler that runs on the computer for which it assembles programs. The self-assembler will require some memory and peripherals, and it may run quite slowly compared to a cross-assembler.

A *macroassembler* is an assembler that allows you to define sequences of instructions as macros.

A *microassembler* is an assembler used to write the microprograms which define the instruction set of a computer. Microprogramming has nothing specifically to do with programming microcomputers, but has to do with the internal operation of the computer.

A *meta-assembler* is an assembler that can handle many different instruction sets. The user must define the particular instruction set being used.

A *one-pass assembler* is an assembler that goes through the assembly language program only once. Such an assembler must have some way of resolving forward references, for example, Jump instructions which use labels that have not yet been defined.

A *two-pass assembler* is an assembler that goes through the assembly language source program twice. The first time the assembler simply collects and defines all the symbols; the second time it replaces the references with the actual definitions. A two-pass assembler has no problems with forward references but may be quite slow if no backup storage (like a floppy disk) is available; then the assembler must physically read the program twice from a slow input medium (like a teletypewriter paper tape reader). Most microprocessor-based assemblers require two passes.

2.7 Errors

Assemblers normally provide error messages, often consisting of an error code number. Some typical errors are:

Undefined name	Often a misspelling or an omitted definition
Illegal character	Such as a 2 in a binary number
Illegal format	A wrong delimiter or incorrect operands
Invalid expression	for example, two operators in a row
Illegal value	Usually the value is too large
Missing operand	Pretty self explanatory
Double definition	Two different values assigned to one name
Illegal label	Such as a label on a pseudo-operation that cannot have one
Missing label	Probably a miss spelt lable name
Undefined operation code	

In interpreting assembler errors, you must remember that the assembler may get on the wrong track if it finds a stray letter, an extra space, or incorrect punctuation. The assembler will then proceed to misinterpret the succeeding instructions and produce meaningless error messages.

Always look at the first error very carefully; subsequent ones may depend on it. Caution and consistent adherence to standard formats will eliminate many annoying mistakes.

2.8 Loaders

The loader is the program which actually takes the output (object code) from the assembler and places it in memory. Loaders range from the very simple to the very complex. We will describe a few different types.

A *bootstrap loader* is a program that uses its own first few instructions to load the rest of itself or another loader program into memory. The bootstrap loader may be in ROM, or you may have to enter it into the computer memory using front panel switches. The assembler may place a bootstrap loader at the start of the object program that it produces.

A *relocating loader* can load programs anywhere in memory. It typically loads each program into the memory space immediately following that used by the previous program. The programs, however, must themselves be capable of being moved around in this way; that is, they must be relocatable. An *absolute loader,* in contrast, will always place the programs in the same area of memory.

A *linking loader* loads programs and subroutines that have been assembled separately; it resolves cross-references — that is, instructions in one program that refer to a label in another program. Object programs loaded by a linking loader must be created by an assembler that allows external references. An alternative approach is to separate the linking and loading functions and have the linking performed by a program called a *link editor* and the loading done by a loader.

3 ARM Architecture

This chapter outlines the ARM processor's architecture and describes the syntax rules of the ARM assembler. Later chapters of this book describe the ARM's stack and exception processing system in more detail.

Figure 3.1 on the following page shows the internal structure of the ARM processor. The ARM is a *Reduced Instruction Set Computer* (RISC) system and includes the attributes typical to that type of system:

- A large array of uniform registers.
- A load/store model of data-processing where operations can only operate on registers and not directly on memory. This requires that all data be loaded into registers before an operation can be preformed, the result can then be used for further processing or stored back into memory.
- A small number of addressing modes with all load/store addresses begin determined from registers and instruction fields only.
- A uniform fixed length instruction (32-bit).

In addition to these traditional features of a RISC system the ARM provides a number of additional features:

- Separate *Arithmetic Logic Unit* (ALU) and shifter giving additional control over data processing to maximize execution speed.
- Auto-increment and Auto-decrement addressing modes to improve the operation of program loops.
- Conditional execution of instructions to reduce pipeline flushing and thus increase execution speed.

3.1 Processor modes

The ARM supports the seven processor modes shown in table 3.1.

Mode changes can be made under software control, or can be caused by external interrupts or exception processing.

Most application programs execute in User mode. While the processor is in User mode, the program being executed is unable to access some protected system resources or to change mode, other than by causing an exception to occur (see 3.4 on page 29). This allows a suitably written operating system to control the use of system resources.

The modes other than User mode are known as *privileged modes*. They have full access to system resources and can change mode freely. Five of them are known as *exception modes*: FIQ (Fast Interrupt), IRQ (Interrupt), Supervisor, Abort, and Undefined. These are entered when specific

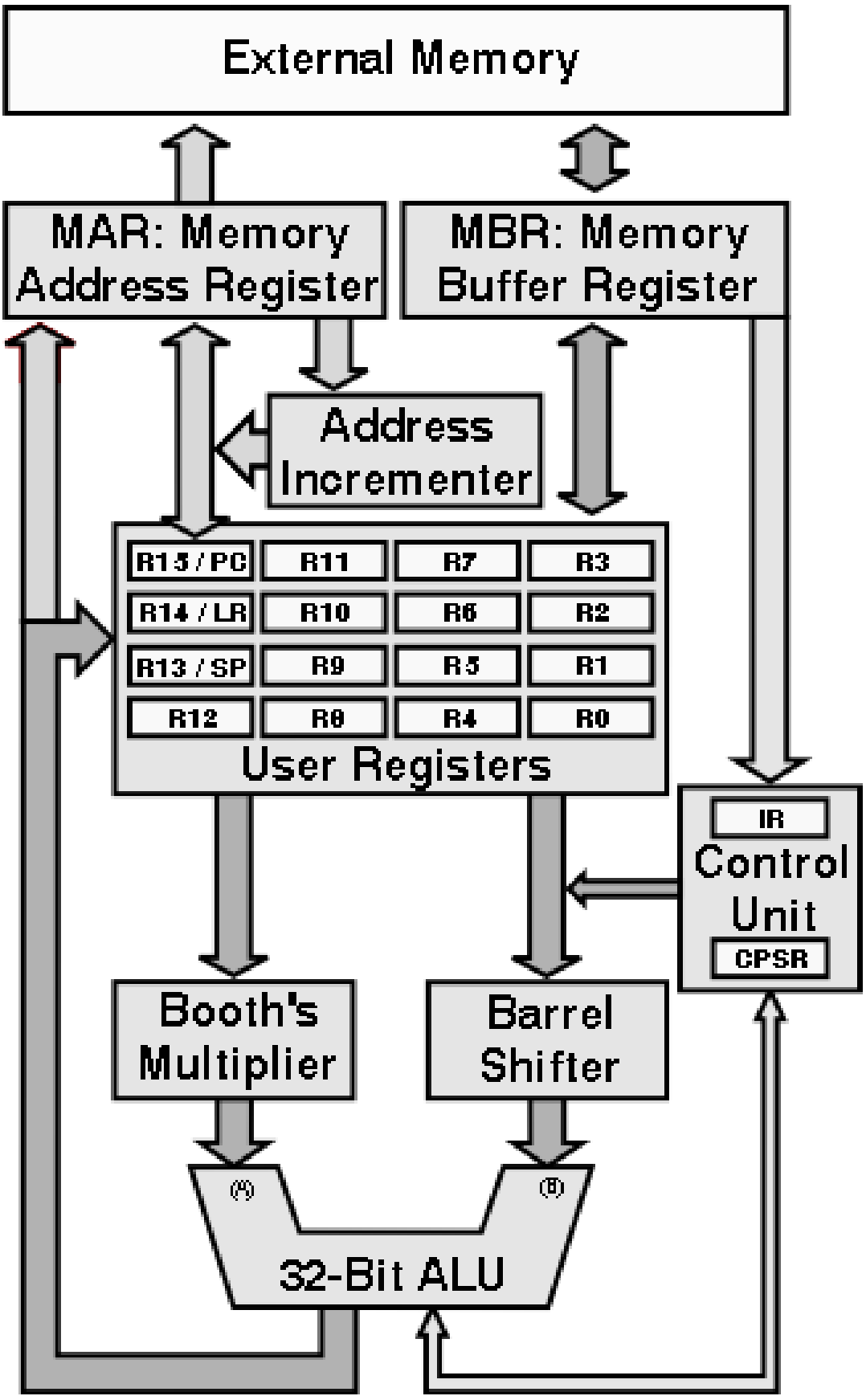


Figure 3.1: ARM Block Diagram

Processor mode		Description
User	usr	Normal program execution mode
FIQ	fiq	Fast Interrupt for high-speed data transfer
IRQ	irq	Used for general-purpose interrupt handling
Supervisor	svc	A protected mode for the operating system
Abort	abt	Implements virtual memory and/or memory protection
Undefined	und	Supports software emulation of hardware coprocessors
System	sys	Runs privileged operating system tasks

Table 3.1: ARM processor modes

exceptions occur. Each of them has some additional registers to avoid corrupting User mode state when the exception occurs (see 3.2 for details).

The remaining mode is System mode, it is not entered by any exception and has exactly the same registers available as User mode. However, it is a privileged mode and is therefore not subject to the User mode restrictions. It is intended for use by operating system tasks which need access to system resources, but wish to avoid using the additional registers associated with the exception modes. Avoiding such use ensures that the task state is not corrupted by the occurrence of any exception.

3.2 Registers

The ARM has a total of 37 registers. These comprise 30 general purpose registers, 6 status registers and a program counter. Figure 3.2 illustrates the registers of the ARM. Only fifteen of the general purpose registers are available at any one time depending on the processor mode.

There are a standard set of eight general purpose registers that are always available (R*0* – R*7*) no matter which mode the processor is in. These registers are truly general-purpose, with no special uses being placed on them by the processors' architecture.

A few registers (R*8* – R*12*) are common to all processor modes with the exception of the fiq mode. This means that to all intent and purpose these are general registers and have no special use. However, when the processor is in the fast interrupt mode these registers and replaced with different set of registers (R*8_fiq* - R*12_fiq*). Although the processor does not give any special purpose to these registers they can be used to hold information between fast interrupts. You can consider they to be `static` registers. The idea is that you can make a fast interrupt even faster by holding information in these registers.

The general purpose registers can be used to handle 8-bit bytes and 32-bit words[1]. When we use a 32-bit register in a byte instruction only the least significant 8 bits are used. Figure 3.3 on the following page demonstrates this.

The remaining registers (R*13* – R*15*) are special purpose registers and have very specific roles: R*13* is also known as the Stack Pointer, while R*14* is known as the Link Register, and R*15* is the Program Counter. The "user" (usr) and "System" (sys) modes share the same registers. The exception modes all have their own version of these registers. Making a reference to register R*14* will assume you are referring to the register for the current processor mode. If you wish to refer to the user mode version of this register you have refer to the R*14_usr* register. You may only refer to register from other modes when the processor is in one of the privileged modes, i.e., any mode other than user mode.

[1] Later revisions of the ARM architecture are also able to handle 16-bit half-words.

Modes						
	Privileged Modes					
		Exception Modes				
User	System	Supervisor	Abort	Undefined	Interrupt	Fast Interrupt
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8	R8_fiq
R9	R9	R9	R9	R9	R9	R9_fiq
R10	R10	R10	R10	R10	R10	R10_fiq
R11	R11	R11	R11	R11	R11	R11_fiq
R12	R12	R12	R12	R12	R12	R12_fiq
R13	R13	R13_svc	R13_abt	R13_und	R13_irq	R13_fiq
R14	R14	R14_svc	R14_abt	R14_und	R14_irq	R14_fiq
PC	PC	PC	PC	PC	PC	PC
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

Figure 3.2: Register Organization

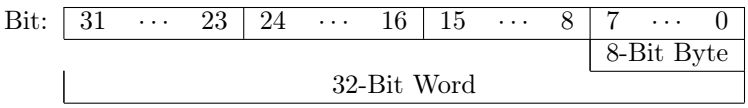


Figure 3.3: Byte/Word

There are also one or two status registers depending on which mode the processor is in. The Current Processor Status Register (CPSR) holds information about the current status of the processor (including its current mode). In the exception modes there is an additional Saved Processor Status Register (SPSR) which holds information on the processors state before the system changed into this mode, i.e., the processor status just before an exception.

3.2.1 The stack pointer, SP or R*13*

Register R*13* is used as a stack pointer and is also known as the SP register. Each exception mode has its own version of R*13*, which points to a stack dedicated to that exception mode.

The stack is typically used to store temporary values. It is normal to store the contents of any registers a function is going to use on the stack on entry to a subroutine. This leaves the register free for use during the function. The routine can then recover the register values from the stack on exit from the subroutine. In this way the subroutine can preserve the value of the register and not corrupt the value as would otherwise be the case.

See Chapter 14 for more information on using the stack.

3.2.2 The Link Register, LR or R*14*

Register R*14* is also known as the *Link Register* or LR.

It is used to hold the return address for a subroutine. When a subroutine call is performed via a BL instruction, R*14* is set to the address of the next instruction. To return from a subroutine you need to copy the Link Register into the Program Counter. This is typically done in one of the two ways:

- Execute either of these instructions:

  ```
  MOV   PC, LR        or        BAL   LR
  ```

- On entry to the subroutine store R*14* to the stack with an instruction of the form:

  ```
  STMIA   SP!,{⟨registers⟩, LR}
  ```

  and use a matching instruction to return from the subroutine:

  ```
  LDMIA   SP!,{⟨registers⟩, PC}
  ```

  This saves the Link Register on the stack at the start of the subroutine. On exit from the subroutine it collects all the values it placed on the stack, including the return address that was in the Link Register, except it returns this address directly into the Program Counter instead.

See Chapter 14 for further details on subroutines and using the stack.

When an exception occurs, the exception mode's version of R*14* is set to the address after the instruction which has just been completed. The SPSR is a copy of the CPSR just before the exception occurred. The return from an exception is performed in a similar way to a subroutine return, but using slightly different instructions to ensure full restoration of the state of the program that was being executed when the exception occurred. See 3.4 on page 29 for more details.

3.2.3 The program counter, PC or R*15*

Register R*15* holds the *Program Counter* known as the PC. It is used to identify which instruction is to be preformed *next*. As the PC holds the address of the next instruction it is often referred to as an *instruction pointer*. The name "program counter" dates back to the times when program instructions where read in off of punched cards, it refers to the card position within a stack of cards. In spite of its name it does not actually count anything!

Reading the program counter

When an instruction reads the PC the value returned is the address of the current instruction plus 8 bytes. This is the address of the instruction *after* the *next* instruction to be executed[2].

This way of reading the PC is primarily used for quick, position-independent addressing of nearby instructions and data, including position-independent branching within a program.

An exception to this rule occurs when an STR (Store Register) or STM (Store Multiple Registers) instruction stores R*15*. The value stored is UNKNOWN and it is best to avoid the use of these instructions that store R*15*.

[2] This is caused by the processor having already fetched the next instruction from memory while it is decoding the current instruction. Thus the PC is still the next instruction to be executed, but that is not the instruction immediately after the current one.

Writing the program counter

When an instruction writes to R*15* the normal result is that the value written is treated as an instruction address and the system starts to execute the instruction at that address[3].

3.2.4 Current Processor Status Registers: CPSR

Rather surprisingly the *current processor status register* (CPSR) contains the current status of the processor. This includes various condition code flags, interrupt status, processor mode and other status and control information.

The exception modes also have a *saved processor status register* (SPSR), that is used to preserve the value of the CPSR when the associated exception occurs. Because the User and System modes are not exception modes, there is no **SPSR** available.

Figure 3.4 shows the format of the CPSR and the SPSR registers.

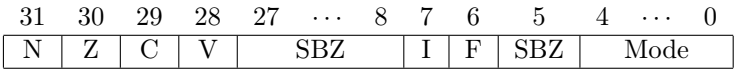


Figure 3.4: Structure of the Processor Status Registers

The processors' status is split into two distinct parts: the User flags and the Systems Control flags. The upper byte is accessible in User mode and contains a set of flags which can be used to effect the operation of a program, see section 3.3. The lower byte contains the System Control information.

Any bit not currently used is reserved for future use and should be zero, and are marked SBZ in the figure. The I and F bits indicate if Interrupts (I) or Fast Interrupts (F) are allowed. The Mode bits indicate which operating mode the processor is in (see 3.1 on page 23).

The system flags can only be altered when the processor is in protected mode. User mode programs can not alter the status register except for the condition code flags.

3.3 Flags

The upper four bits of the status register contains a set of four flags, collectively known at the *condition code*. The condition code can be used to control the flow of the program execution. The is often abbreviated to just ⟨*cc*⟩. The condition code flags are:

`N` The Negative (sign) flag takes on the value of the most significant bit of a result. Thus when an operation produces a negative result the negative flag is set and a positive result results in a the negative flag being reset. This assumes the values are in standard two's complement form. If the values are unsigned the negative flag can be ignored or used to identify the value of the most significant bit of the result.

`Z` The Zero flag is set when an operation produces a zero result. It is reset when an operation produces a non-zero result.

`C` The Carry flag holds the carry from the most significant bit produced by arithmetic operations or shifts. As with most processors, the carry flag is inverted after a subtraction so that the flag acts as a borrow flag after a subtraction.

[3] As the processor has already fetched the instruction after the current instruction it is required to flush the instruction cache and start again. This will cause a short, but not significant, delay.

`V` The Overflow flag is set when an arithmetic result is greater than can be represented in a register.

Many instructions can modify the flags, these include comparison, arithmetic, logical and move instructions. Most of the instructions have an `S` qualifier which instructs the processor to set the condition code flags or not.

3.4 Exceptions

Exceptions are generated by internal and external sources to cause the processor to handle an event, such as an externally generated interrupt or an attempt to execute an undefined instruction. The ARM supports seven types of exception, and a provides a privileged processing mode for each type. Table 3.2 lists the type of exception and the processor mode associated with it.

When an exception occurs, some of the standard registers are replaced with registers specific to the exception mode. All exception modes have their own Stack Pointer (SP) and Link (LR) registers. The fast interrupt mode has more registers (R*8_fiq* – R*12_fiq*) for fast interrupt processing.

Exception Type	**Processor Mode**	
Reset	Supervisor	svc
Software Interrupt	Supervisor	svc
Undefined Instruction	Undefined	und
Prefetch Abort	Abort	abt
Data Abort	Abort	abt
Interrupt	IRQ	irq
Fast Interrupt	FIQ	fiq

Table 3.2: Exception processing modes

The seven exceptions are:

Reset when the Reset pin is held low, this is normally when the system is first turned on or when the reset button is pressed.

Software Interrupt is generally used to allow user mode programs to call the operating system. The user program executes a software interrupt (`SWI`, A.26 on page 152) instruction with a argument which identifies the function the user wishes to preform.

Undefined Instruction is when an attempt is made to preform an undefined instruction. This normally happens when there is a logical error in the program and the processor starts to execute data rather than program code.

Prefetch Abort occurs when the processor attempts to access memory that does not exist.

Data Abort occurs when attempting to access a word on a non-word aligned boundary. The lower two bits of a memory must be zero when accessing a word.

Interrupt occurs when an external device asserts the IRQ (interrupt) pin on the processor. This can be used by external devices to request attention from the processor. An interrupt can not be interrupted with the exception of a fast interrupt.

Fast Interrupt occurs when an external device asserts the FIQ (fast interrupt) pin. This is designed to support data transfer and has sufficient private registers to remove the need for register saving in such applications. A fast interrupt can not be interrupted.

When an exception occurs, the processor halts execution after the current instruction. The state of the processor is preserved in the *Saved Processor Status Register* (SPSR) so that the original

program can be resumed when the exception routine has completed. The address of the instruction the processor was just about to execute is placed into the Link Register of the appropriate processor mode. The processor is now ready to begin execution of the exception handler.

The exception handler are located a pre-defined locations known as *exception vectors*. It is the responsibility of an operating system to provide suitable exception handling.

3.5 Register Transfer Language

Before continuing, we need to develop an unambiguous notation to help us describe the way in which information moves around the processor (see figure 3.1 on page 24). The *register transfer language* (RTL) is just such a notation.

Each component of the processor is given a name or an abbreviation, for example the Memory Address Register is known as the MAR, and the Program Counter is refereed to as PC. A left-, or back-arrow (←) indicates the transfer of data from one component to another. Thus the RTL expression:

MAR ← PC

means that the contents of the program counter are transferred (i.e. copied into) the memory address register. A comment can be added to the line by placing the text after a semi-colon (;) following the expression.

In addition to accessing a component directly we can also refer to a particular field, or part, of a device by placing the name of the field in parentheses after the device name. For example, the Instruction Register (IR) is split into a number of fields including the operation code (or op-code) field (see section 3.6.2 for a further description of the IR fields). In order to access the op-code field we would need to write:

IR(op-code)

A field is not always given a name, so we need to indicate the field by specifying which bits have been grouped to provide the field. This is known as a *bit field* which we denote by giving the lower and upper bits, separated by a colon as the field name. Thus to select the upper four bits (bits 28 to 31 inclusive) of register R*4* we would write:

R*4*(28:31)

Finally, we also have the notion of a guard. This is a condition which must be true before the expression can be evaluated. The guard is written before the RTL expression it is guarding and is separated from that expression with a colon (:). Normally the guard is a test for an optional item in the instruction. For example, there is a version of the `MOV` instruction (`MOVS`) which sets the CPSR flags N and Z. We can place a guard, which test for the extra `S` like this:

⟨*S*⟩: CPSR ← ALU(Flags)

When more than one guard is required we simply list the guards next to each other in sequence:

⟨*cc*⟩⟨*S*⟩: CPSR ← ALU(Flags)

indicates the process must be in the current condition, as indicated by the ⟨*cc*⟩ guard (see section 4.1.2 on page 42), and it must be the `S` form of the instruction before the RTL expression is be executed.

3.5.1 Memory

When accessing external random access memory the processor must go though the memory device, indicated by the device name M. The processor must first place the address, or location, it intends to access in the Memory Address Register (MAR). When writing to external memory the value to be written should then be copied into the Memory Buffer Register (MBR). When reading from memory the value made available in the MBR.

The following RTL demonstrates how the system accesses memory. The location we wish to access is held in the register R*12*. In the first example we are reading the value from memory into register R*0* while the second example writes the value in register R*1* to memory.

Read	Write
MAR ← R*12*	MAR ← R*12*
MBR ← M(MAR)	MBR ← R*1*
R*0* ← MBR	M(MAR) ← MBR

In particular you should note the two lines which include the item M(MAR). This is where the data is actually transferred between the processor and external memory.

Although this is the correct way of accessing external memory, it is rather tiresome. To overcome this we abuse the notation slightly by placing the location we wish to access as a field to the memory device, and read/write to it directly. Thus we can write the above examples as:

$$\mathsf{R}0 \leftarrow \mathsf{M}(\mathsf{R}12) \qquad \mathsf{M}(\mathsf{R}12) \leftarrow \mathsf{R}1$$

In this way we hide the reference to the MAR and MBR inside the reference to the memory device M(...).

3.5.2 Arithmetic and Logic Unit

In a similar manner the ALU has a number of rules for its use. The ALU has a number of parts, or registers. Normally the ALU requires two operands (arguments) and a command.

A — The first operand (or argument) goes into ALU register A, written as ALU(A). The design of the processor means that only a register may be moved into the A register.

B — The second operand (or argument) should be placed in the ALU's B register, denoted by ALU(B). The value for the B register may come from a number of different sources, normally a register. As the **B** register is connected to the Barrel Shift component the value can be modified (shifted) as it is copied into the B register, this is discussed later.

Cmd — Once the two operands have been set up, copied into registers A and B, the ALU needs to know what to do with them. Thus we also have a Command (Cmd) register. We should write this as ALU(Cmd) but we normally miss off the register name when writing to the ALU.

The commands the ALU can process are:

add	$\mathsf{R} = \mathsf{A} + \mathsf{B}$	Add A to B
subtract	$\mathsf{R} = \mathsf{A} - \mathsf{B}$	Subtract B from A
and	$\mathsf{R} = \mathsf{A} \wedge \mathsf{B}$	Bitwise AND of A and B
or	$\mathsf{R} = \mathsf{A} \vee \mathsf{B}$	Bitwise OR of A and B
eor	$\mathsf{R} = \mathsf{A} \oplus \mathsf{B}$	Exclusive OR of A and B
not	$\mathsf{R} = \overline{\mathsf{B}}$	Logical complement of B

R — Having preformed some operation the result of that operation is placed in the Result (R) register. As with the command register we should refer to this register as ALU(R),

however, when reading from the ALU it can be assumed we are referring to the Result register, so we miss it off.

Flags — Finally we have the flags register. This includes the N, Z, C and V flags after the operation. The state of these flags can be copied into the Current Process Status Register (CPSR) for use with conditional execution, discussed in section 4.1.2 on page 42.

To demonstrate the way the ALU works, let us look at how it would add two registers together. The instruction `ADD R0, R1, R2` would add the content of register R*1* to the content of register R*2* placing the result in the register R*0*. The register transfer language to describe this would be:

$$\begin{aligned} \text{ALU(A)} &\leftarrow \text{R}1 && ; \textit{First operand} \\ \text{ALU(B)} &\leftarrow \text{R}2 && ; \textit{Second operand} \\ \text{ALU} &\leftarrow \text{add} && ; \textit{ALU command} \\ \text{R}0 &\leftarrow \text{ALU} && ; \textit{Read result} \end{aligned}$$

Note the lack of field in the last two lines. We are using the defaults for the write (cmd) and read (R) operations. As with the memory operations we tend to abuse the notation by writing a single line which summarised this operation:

$$\text{R}0 \leftarrow \text{R}1 + \text{R}2$$

The convention is that the item before the operator is placed in the ALU register A and the item after the operator is placed in register B. In our example the operator is the plus sign (+), while the register R*1* would be copied into ALU(A) and R*2* would be copied into ALU(B).

The data paths leading to the A and B registers pass though two additional components that preform operations which traditionally form part of the Arithmetic and Logic Unit. These are the Booth Multiplier and Barrel Shifter respectively.

Booth Multiplier

The Booth multiplier is a hardware component that is capable of multiplying two signed numbers. It can take two 16-bit values and produces a single 32-bit result. As with the ALU, the Booth Multiplier (BM) has two input registers (A and B) and one results register (R). The instruction:

```
MUL    R0, R1, R2
```

will multiply the value in register R*1* with that in register R*2*, placing the result in R*0*. The RTL for this would be:

$$\begin{aligned} \text{BM(A)} &\leftarrow \text{R}1(0{:}15) && ; \textit{First operand} \\ \text{BM(B)} &\leftarrow \text{R}2(0{:}15) && ; \textit{Second operand} \\ \text{ALU(A)} &\leftarrow \text{BM(R)} && ; \textit{Result gos on to the ALU} \\ \text{R}0 &\leftarrow \text{ALU} && ; \textit{Read result} \end{aligned}$$

There are two points to note here, firstly only the lower 16-bits (or halfwords) of the operands are used in the multiply operation. The second point to note is that the result must go on to the ALU before it can be copied into the destination register.

You may be surprised to discover that whilst this is what actually happens inside the processor we tend to write it differently:

$$\text{R}0 \leftarrow \text{R}1 \times \text{R}2$$

The Booth Multiplier is named after Andrew D. Booth who first suggested a method of multiplying two numbers together that could be implemented as a hardware component[4]. See Chapter 12 for a discussion of the Multiply instructions.

[4]Booth's paper *A signed binary multiplication technique* was first published in the *Quarterly Journal of Mechanics and Applied Mathematics*, 4:2 (236–240) in 1951.

Shift Method	RTL	Section	Page
Logical Shift Left	$A \ll B$	5.1.2	51
Logical Shift Right	$A \gg B$	5.1.3	52
Arithmetic Shift Right	$A \Rightarrow\!\!\gg B$	5.1.4	52
Rotate Right	$A \ggg B$	5.1.5	53
Rotate Right Extend	$A \ggg B$	5.1.6	54

Table 3.3: Barrel Shifter Operations

Barrel Shifter

The Barrel Shift unit allows the ARM to manipulate the second operand, leading to register B of the ALU, before it actually reaches it. This is an advantage when dealing with bit-orientated operations (Chapter 8) and data structures (Chapter 13). For a discussion on the use of the Barrel Shift we refer you to Chapter 5. Particularly the discussion of the data addressing mode ($\langle op1 \rangle$) in section 5.1 on page 51.

Table 3.3 above shows how we write the five different ways in which data can be manipulated by the barrel shift, and which section of Chapter 5 discusses the shift method.

For example, if we wish to add the value of register R2 shifted left by 4 bits (effectively multiplied by 16) to the register R1 leaving the result in register R0, we would give the instruction:

```
ADD    R0, R1, R2, LSL #4
```

Which would be represented in RTL as:

$$R0 \leftarrow R1 + R2 \ll 4$$

Note the use of the shorthand form of the ALU operation + (add). As the shift is on the right hand side of the operator, the result must be placed in register B of the ALU. It is only data going to register B which can be shifted in the manner anyhow so that works out.

You should also note that most processes require a separate instruction to preform these operations.

3.6 Control Unit

The Control Unit is the most complex part of the processor. This controls the overall operation of the processor. It sends control signals to the other devices prompting them to place data on one of the buses or take data from the bus. The control unit is the device which actually executes the RTL we have been looking at. Indeed the purpose of this book is to describe the operation of this unit.

So what happens in the Control Unit? In essence it is quite simple, it reads a machine instruction from memory and then preforms the operation described by the instruction. It does this though the now famous fetch/execute cycle. This starts with the fetch phase where the control unit will fetch the next instruction from memory into the Instruction Register (IR).

In older microprocessors the execute phase was a simple task. When processors become more complex a micro-code system was introduced where the execute phase consisted of executing a series of RTL like instructions within the control unit itself. Such systems are refereed to as Complex Instruction Set Computers (CISC). Such systems include the Motorola MC68000, and the Intel 80x86 series.

The ARM however, is a Reduced Instruction Set Computer (RISC) system, which means the designers chose to use a larger instruction size (32-bit) in exchange for making the control unit simple, well simpler. Other RISC processors include the SPARC, and the PowerPC range (Gn).

A RISC processor has a number of stages to the execute part of the fetch/execute cycle:

Instruction Fetch	Fetch the instruction from memory into the instruction register.
Instruction Decode	Decode the instruction in the instruction register working out what we are supposed to do next.
Operand Fetch	Fetch the source operands for the task, this may involve the use of the data addressing mode (see 5.1 on page 51) or a memory addressing mode (see 5.2 on page 54).
Execute	Perform the requested operation.
Operand Store	Save the result someplace, this will almost always be a register.

The ARM have been designed to perform these stages simultaneously. So whilst it is decoding one instruction it can be fetching the next instruction from memory. This is known as the *instruction pipeline*.

The way in which the pipeline works can best be seen by examining the RTL the control unit will generate when processing the instruction

```
SUBS   R0, R0, #1
```

This instruction will subtract 1 from the content of the register R*0*, and set the Z flag should it become zero. This is a particularly useful instruction as we will see in chapter 9.

It should be noted that while we discuss a five stage instruction pipeline, different variants of the ARM have a different stages. The version of the ARM we are using has a three stage pipeline: Instruction Fetch and Decode; Operand Fetch; Execute and Operand Store.

3.6.1 Instruction Fetch

The first step is to copy the memory location contained in the program counter into the memory address register.

$$\mathsf{MAR} \leftarrow \mathsf{PC}$$

The program counter is badly named as it does not count programs, or anything else for that matter, but contains the address of the next instruction in memory to be executed. In some system it is called the instruction pointer, or IP. Once the MAR has the location of the next instruction, the contents of the program counter are incremented (moved on to the next instruction) with a special address incrementer (INC) circuit and moved back to the program counter. In this way, the program counter is pointing to the next instruction while the current instruction is being executed.

$$\begin{aligned} \mathsf{INC} &\leftarrow \mathsf{MAR} \\ \mathsf{PC} &\leftarrow \mathsf{INC} \end{aligned}$$

The MAR now contains a copy of the contents of the PC, so that the instruction to be executed is read from the memory and transferred to the memory buffer register (MBR). Once in the MBR the instruction can then be copied into the instruction register (IR).

$$\begin{aligned} \mathsf{MBR} &\leftarrow \mathsf{M}(\mathsf{MAR}) \\ \mathsf{IR} &\leftarrow \mathsf{MBR} \end{aligned}$$

By abusing the notation we can reduce these five lines to just two:

```
IR ← M(PC)     ; Read value at PC into the Instruction Register
PC ← INC(PC)   ; Move Program Counter to next instruction
```

3.6.2 Instruction Decode

Now the instruction is in the instruction register it is necessary to decode it. The IR is divided into a number of fields or parts.

op-code
This is the operation code or binary instruction which tells the control unit which function to preform.

condition code.
This is a general guard for the whole instruction. The instruction will only be preformed if the process is in this state or the condition code has been set to *always*, the default. See section 4.1.2 on page 42 for more information. Our example instruction has the default setting, so there is not guard and the instruction will always be executed.

Set flags.
Most of the data processing instructions include a `S` variation. Such as our `SUBS` instruction. The ⟨*S*⟩ field used to indicate whether the instruction should set the CPSR flags (as in our example) or not.

destination register.
All of the instructions take one or two source values, and preform some operation on them, placing a result in the destination register. With the exception of the store instruction it is always a register. In our example this would be the register R*0*.

source register.
The majority of instructions require one or two source values to operate on. The ⟨*source*⟩ register indicates which register holds the source value for the instruction/operation. In our example this would also be the register R*0*.

op1 The data processing instructions all require a source value which can be calculated as part of the instruction. This is known as a ⟨*op1*⟩ value. See section 5.1 on page 51 for a full discussion of the possible values for ⟨*op1*⟩. Our subtract instruction is a data processing instruction and ⟨*op1*⟩ is an immediate value, so it will take the value from the ⟨*value*⟩ part of the instruction register.

op2 The memory based instructions require a memory location to work with, this is specified in an effective address, known as an ⟨*op2*⟩ operand. Section 5.2 on page 54 goes into the details of the ⟨*op2*⟩ field.

value
Holds a small value as part of the binary instruction. This is normally used when calculating the second operand, either ⟨*op1*⟩ or ⟨*op2*⟩. As we are using immediate addressing in our example instruction (the `#1`), the value (1) will be in the ⟨*value*⟩ field.

offset
This is similar to ⟨*value*⟩ except the value is larger. This field is only used by the branch instructions.

Whilst all of the fields are available, they only have any real meaning in the context of the instruction. The only fields that have any real value are the op-code and condition fields. Even then not all instructions have an condition field.

The instruction decode stage of the fetch/execute cycle does not actually produce any RTL. However, for our example instruction the IR will be broken down as follows:

op-code	condition	set	destination	source	op1	value
subtract	*always*	*true*	R*0*	R*0*	*immediate*	*1*

3.6.3 Operand Fetch

The operand fetch stage will prepare the ALU for the execution phase by reading the operands into the ALU registers.

In our example we read the ⟨*source*⟩ register into the ALU's A register.

ALU(A) ← R*0*

At the same time we can also copy the second operand into register B of the ALU. As this is a data processing instruction the control unit will analyse IR(op1) and see that we are using an immediate value. Thus it will copy the *value* field into the ALU register.

ALU(B) ← IR(value)

As the ⟨*op1*⟩ value passes through the Barrel Shifter we can preform a shift operation on the value as it passes into the ALU.

For the memory load instruction, the operand fetch stage will read a value from external memory as specified by ⟨*op2*⟩.

3.6.4 Execute

Now that the ALU has been configured, both registers have been loaded with the appropriate operands (values), we can now instruct the ALU to subtract the second operand from the first. This is done by simply sending a *subtract* message to the ALU.

ALU ← subtract

3.6.5 Operand Store

Finally we need to store the result of this operation by copying the value from the ALU to the destination register.

R*0* ← ALU

However we also have the ⟨*S*⟩ flag set, so we must copy the flags from the ALU back into the CPSR in the control unit.

CPSR ← ALU(flags)

For the memory store instruction the destination is a memory location specified by ⟨*op2*⟩. This is the only instruction that does not use a register as the destination. This will cause the system to write the result to memory.

3.6.6 Summary

We have just looked at the fetch/execute cycle and the way the system actually processes an instruction. In particular we looked at the processing of a specific instruction

```
SUBS   R0, R0, #1
```

which subtracts 1 from the content of the register R*0*, and sets the Z flag should it become zero.

Here we will list the RTL that was produced without all the bothersome interpretation.

```
MAR ← PC          ; Instruction Fetch
INC ← MAR
 PC ← INC
MBR ← M(MAR)
 IR ← MBR
                  ; Instruction Decode
ALU(A) ← R0       ; Operand Fetch
ALU(B) ← IR(value)
 ALU ← subtract   ; Execute the instruction
 R0 ← ALU         ; Operand Store
CPSR ← ALU(flags)
```

If we abuse the notation, as discussed earlier in sections 3.5.2 and 3.6.1, we can reduce this from 10 instructions to just four.

```
 IR ← M(PC)          ; Fetch the instruction
 PC ← INC(PC)        ; Move on to the next instruction
 R0 ← R0 − IR(value) ; Execute the subtract
CSPR ← ALU(flags)    ; Save the flags
```

Although this might be smaller and easier to read. Unfortunately it does not describe the data flow correctly, the longer version is more correct.

4 Instruction Set

Why are a microprocessor's instructions referred to as an instruction set? Because the microprocessor designer selects the instruction complement with great care; it must be easy to execute complex operations as a sequence of simple events, each of which is represented by one instruction from a well-designed instruction set.

Assembler often frighten users who are new to programming. Yet taken in isolation, the operations involved in the execution of a single instruction are usually easy to follow. Furthermore, you need not attempt to understand all the instructions at once. As you study each of the programs in this book you will learn about the specific instructions involved.

From the advent of the microprocessor in the early 1970s there has been a trend for instruction sets to become more sophisticated and complex. In the early 1980s, a different approach to the instruction set emerged, that of the Reduced Instruction Set Computer or RISC, which attempted to produce a 'striped down' processor, or a 'supercharged' one if you prefer.

When David Patterson and David Ditzel[1] first proposed the idea, they analysed a large number of programs to find out how developers where actually using the processor. They came up with three interesting observations. The first of which can be seen in table 4.1 below.

Instruction Grouping		Usage	Instruction Grouping		Usage
1	Data Movement	45.28%	*5*	Logical	3.91%
2	Flow Control	28.73%	*6*	Shift	2.93%
3	Arithmetic	10.75%	*7*	Bit Manipulation	2.04%
4	Comparison	5.92%	*8*	Input/Output and Miscellaneous	0.44%

Table 4.1: Instruction Group Average Usage

This shows that most programs spend over 80% of the time executing instructions from the Data Movement, Flow Control or Arithmetic instruction groups. If they could find a method of increasing the speed of these instructions, the program would execute that much faster.

There second observation was in the use of constant values. Some 56% of constant values where within the range of ±15, while 98% of constants where in the range ±511. As the most constant values are small enough (just 5 bits) they could be included as part of the instruction, rather than forcing yet another memory access for it.

The final observation was that the number of parameters (or arguments) passed between subroutines was normally less than six. Remember this research was done before the advent of Microsoft Windows and the 13-parameter API call which nobody can ever remember. Thus if the processor had sufficient registers, it would not be necessary to use external (off-chip) memory for a stack frame. See Chapter **??** for a discussion of the stack frame.

[1] David A. Patterson and David R. Ditzel, *The case for the reduced instruction set computer* published in *Computer Architecture News*, volume 9(3) in 1980.

Mnemonic	Instruction	Mnemonic	Instruction
	Data Movement		
`MOV`	Move		
	Arithmetic		
`ADD`	Add	`ADC`	Add with Carry
`SUB`	Subtract	`SBC`	Subtract with Carry
`RSB`	Reverse Subtract	`RSC`	Reverse Subtract with Carry
`MUL`	Multiply	`MLA`	Multiply Accumulate
	Flow Control		
`CMP`	Compare	`CMN`	Compare Negative
`TEQ`	Test Equivalence	`TST`	Test
`BAL`	Branch Always	`B`$\langle cc \rangle$	Conditional Branch
`BLAL`	Branch and Link Always	`BL`$\langle cc \rangle$	Conditional Branch and Link
	Memory Access		
`LDR`	Load Register	`STR`	Store Register
`LDRB`	Load Register Byte	`STRB`	Store Register Byte
`LDM`	Load Multiple	`STM`	Store Multiple
	Logical and Bit Manipulation		
`AND`	Bitwise AND	`BIC`	Bit Clear
`ORR`	Bitwise (inclusive) OR	`EOR`	Bitwise Exclusive OR
`MVN`	Move Negative		
	System Control		
`MRS`	Move to Register from Status	`MSR`	Move to Status from Register
`SWP`	Swap	`SWPB`	Swap Byte
`SWI`	Software Interrupt		

Table 4.2: Instruction Mnemonics

These three observations form the basis of what has become know as the Berkeley RISC architecture, and subsequently formed the principles upon which the ARM instruction set was designed.

The mnemonics for the ARM instruction set are listed in table 4.2 above. This provides a survey of the processors capabilities, and will also be useful when you need a certain kind of operation but are either unsure of the specific mnemonics or not yet familiar with what instructions are available.

The instruction mnemonics are supposed to provide you with an easy way to remember the instructions set. Although, despite having developed in assembler on ten different microprocessors, we have yet to find a set of mnemonics which are truly mnemonic. This is one of the reasons people new to assembler find it so daunting, attempting to remember all of these so called mnemonics. It is probably easier to remember the instruction you want, and then work out the mnemonic for it.

There are generally three types of developer. One who can understand what each instruction does but is not able to put them together to form a program, chapters 7 through to 14 are intended to help these people. The second is one who can understand the principles behind the instructions and is able to put it all together to form a program, but can never remember the precise details of the instructions, appendix A is intended for these people. The final type, is the most annoying, those who can not only put it all together in the form of a program, in addition they are also able to remember each and every instruction in minute detail.

The ARM instruction set can be divided into six broad classes of instruction.

- Data Movement
- Arithmetic
- Flow Control
- Memory Access
- Logical and Bit Manipulation
- System Control

4.1 Data Movement

There is only one instruction which falls into this category, which is the `MOV` (Move) Instruction. This is however probably the most important instruction, as over 40% of a program will consists of this instructions. It is worth our while looking at this instruction in more detail as it shares two attributes with most of the other instructions, namely the *set flags* variant and *conditional execution*.

Section A.13 on page 144 gives the syntax for this instruction as:

`MOV`⟨*cc*⟩⟨*S*⟩ R*d*, ⟨*op1*⟩

which we interpret as:

`MOV` this is the mnemonic for the instruction which indicates which operation we would like to preform. See table 4.2 on the preceding page for a list of mnemonics, and Appendix A for a detailed discussion of each instruction. The `MOV` mnemoic is a bit of a misnomer as the original value is not destroyed, but copied.

⟨*cc*⟩ indicates conditional execution, where the instruction is only executed under the specified condition. This is either a two letter condition code (see table 4.3 on the following page) or missing altogether, in which case it is assumed the instruction should always be executed. This is discussed in more detail later (in section 4.1.2).

⟨*S*⟩ indicates the operation should set the condition code. This is to be used in conjunction with the conditional execution and is discussed in more detail later.

R*d* indicates the *destination* register. This is always a register and may be any register available in the current processor mode.

⟨*op1*⟩ indicates the *source* of the data. The `MOV` instruction is unusual in that it only has the one source. Most instructions require two operands, thus they require two sources. The first is normally a source register (R*s*).

The source register (if present) is copied into the register `A` of the ALU. The ⟨*op1*⟩ source is copied into register `B` of the ALU, which means it has to pass though the Barrel Shifter. This allows the programmer to preform an arithmetic or logical shift of the data en route. As a result there is no requirement for the dedicated shift instructions found in other (CISC) processors. See section 5.1 for a full discussion of what can be done with an ⟨*op1*⟩ value.

With this in mind, the register transfer language notation describing a general `MOV` instruction would be:

$$\begin{aligned} \langle cc\rangle: &\quad \mathsf{ALU(B)} \leftarrow \langle op1\rangle \\ \langle cc\rangle: &\quad \mathsf{R}d \leftarrow \mathsf{ALU} \\ \langle cc\rangle\langle S\rangle: &\quad \mathsf{CSPR} \leftarrow \mathsf{ALU(flags)} \end{aligned}$$

Where the ⟨*cc*⟩ guard indicates the conditional execution, and the ⟨*S*⟩ guard indicates the Set flags variant of the instruction.

4.1.1 Set Flags Variant

Most of the data processing instructions, this includes the Arithmetic, and Logic instruction groups, have a variant which allows them to sets the condition code flags. This variant is indicated by an ⟨*S*⟩ in the syntax definition of the instruction (in Appendix A) and by adding the letter `S` to the end of the mnemonic.

General		*Signed*		*Unsigned*	
CS	*C*arry *S*et	GT	*G*reater *T*han	HI	*Hi*gher Than
CC	*C*arry *C*lear	GE	*G*reater Than or *E*qual	HS	*H*igher or *S*ame (aka CS)
EQ	*Eq*ual (Zero Set)	LT	*L*ess *T*han	LO	*Lo*wer Than
NE	*N*ot *E*qual (Zero Clear)	LE	*L*ess Than or *E*qual	LS	*L*ower or *S*ame (aka CC)
VS	O*v*erflow *S*et	MI	*Mi*nus (Negative)		
VC	O*v*erflow *C*lear	PL	*Pl*us (Positive)		

Table 4.3: ⟨*cc*⟩ (Condition code) Mnemonics

The flow control instructions will always effect the processor status, thus they do not need a variant. The Memory Access instructions do not pass the data via the ALU and so can not set the flags. This is quite a pain, as it means we have to add an extra instruction to check if the value we have just read from memory is a zero or not.

The MOVS instruction will pass the value though the ALU in order to generate the condition codes. In particular it can only set the Zero and Negative flags. This variant can be combined with the flow control and conditional execution feature to provide fast efficient code. Chapter 9 provides a number of examples.

We can indicate the action taken by a *Set Flags* variant by using the ⟨*S*⟩ guard in the RTL description of the instruction:

⟨*S*⟩: CSPR ← ALU(flags)

4.1.2 Conditional Execution: ⟨*cc*⟩

The vast majority of instructions include the condition code (⟨*cc*⟩) attribute which allows the instruction to be executed conditionally dependent on the current status of the processor (condition code flags, see section 3.3 on page 28). If the flags indicate that the condition is true, the instruction is executed, otherwise the instruction is ignored, and the processor simply moves on to the next instruction.

A list of the two letter condition codes are shown in table 4.3 above. To indicate that an instruction is conditional we simply place the two letter mnemonic for the condition after the mnemonic for the instruction, but before the ⟨*S*⟩ if present. If no condition code mnemonic is used the instruction will always be executed, there is even a two letter mnemonic to indicate *always* (AL) if you prefer.

Note that the *Greater* and the *Less* conditions are for use with signed numbers while the *Higher* and *Lower* conditions are for use with unsigned numbers. These condition codes only really make sense after a comparison instruction (see 4.3).

For example, the instruction

```
MOVCC  R0, R1
```

will copy the value in the register R*1* into the R*0* register, only when the Carry flag is clear (or not set, this is the CC condition code), R*0* will remain unaffected if the C flag is set.

Conditional execution can be combined with the ⟨*S*⟩ variant instruction. For example, what would the following code fragment do?

```
MOVS    R0, R1
MOVEQS  R0, R2
MOVEQ   R0, R3
```

1. The first instruction will always copy R*1* into R*0*, but it will also set the N and Z flags accordingly.

2. The second instruction is only executed if the Z flag is set, i.e., the value of R*1* was zero. If the value of R*1* was not zero the instruction is skipped. If the second instruction is executed it will copy the value of R*2* into R*0* and it will also set the N and Z flags according to the value of R*2*.
3. The third instruction is only executed if both R*1* and R*2* are both zero.

4.2 Arithmetic

The eight instructions in this group all preform an arithmetic operation on two (and in one case three) source operands, one of which must be a register (R*n*) and another value ($\langle op1 \rangle$) producing a result which is placed in a destination register (R*d*). They can also optionally update the condition code flags based on the result.

4.2.1 Addition

There are only two addition instructions: `ADD` (Add) and `ADC` (Add with Carry). These instructions both add two 32-bit numbers, but they can be combined to add 64-bit or larger numbers. Let registers R*0*, R*1* and R*2*, R*3* contain two 64-bit numbers, with the lower word in registers R*0* and R*2* respectively. We can add these 64-bit numbers to produce a new 64-bit number in R*4*, R*5*:

```
ADDS   R4, R0, R2   ; Add lower words
ADC    R5, R1, R3   ; Add upper words
```

The `ADDS` not only adds the two lower words together, but also sets the carry flag if there is a carry over, and clears it if there is no carry over. The `ADC` not only adds the upper words, but also adds in the carry over from the lower words.

4.2.2 Subtraction

There are four subtraction instructions. The `SUB` (Subtract) and `SBC` (Subtract with Carry) instructions are the equivalent of the `ADD` and `ADC` instructions above.

Unlike addition, subtraction is not commutative. This means that while the result of adding $4+2$ is the same as that of adding $2+4$, the result of subtracting $4-2$ (2) is not the same as subtracting $2-4$ (-2). To allow for this we have two additional "reverse" instructions: `RSB` (Reverse Subtract) and `RSC` (Reverse Subtract with Carry) which subtract the value in a register from another value.

In subtraction, the carry flag records the fact that a subtraction had to borrow from the next (32^{nd}) bit. Thus the `SUB` and `SBC` instructions can be combined in the same way as the `ADD` and `ADC` instructions to form a 64-bit subtraction. The `RSB` and `RSC` instruction can also be combined in this manner.

4.2.3 Multiplication

There are two multiplication instructions, `MUL` (Multiply) and `MLA` (Multiply Accumulate). These instructions are different from the others in this group, in that they can only operate on registers. The second operand must be a register, rather than an $\langle op1 \rangle$ value, as used by the other instructions in the arithmetic group.

The `MUL` instruction will use the Booth Multiplier to simply multiply two 32-bit numbers, giving a 32-bit result. This causes a small problem, as multiplying two 32-bit numbers should produces

a 64-bit result. So the multiply instructions should be used with caution. It does, however, mean that we can multiply to signed 16-bit numbers, and receive a correct signed 32-bit result.

The `MLA` instruction the same as the `MUL` instruction with the exception that it will add the result of the multiplication to the value in a third register. This can be used for calculating running totals.

4.2.4 Division

If you have been paying attention, you will have realised that we have already discussed the eight instructions in this group. This means that there are no instructions to support division.

Division is performed considerably less frequently than the other arithmetic operations. It is, however, possible to perform division by repeatedly subtracting the *divisor* from the *dividend* until the result is either zero or less than the divisor. The number of times the divisor is subtracted is known as the *quotient*, and the value left after the final subtraction is the *remainder*. That is:

$$dividend = quotient \times divisor + remainder$$

Let us consider the following equation:

$$\frac{10}{7} = 1 \texttt{ remainder } 3$$

in this equation the dividend is 10, the divisor is 7, the result is in two parts the quotient (1) and the remainder (3). This is all very well, but what happens when the numbers are signed, do we use floored or symmetric division?

Floored division is where the quotient is rounded down toward negative infinity and the remainder is used to correct the rounding. Thus if we divide -10 by 7, we would have a quotient of -2 and a remainder of 4 ($7 \times -2 = -14 + 4 = -10$). In symmetric division the rounding is done the other way, towards zero. Thus -10 divided by 7 would have a quotient of -1 with a remainder of -3 ($7 \times -1 = -7 + -3 = -10$).

Just to complicate matter more, like subtraction, division is not commutative. Given that division is the least used arithmetic operation is it any wonder there are no instructions to support (integer) division.

4.3 Flow Control

The flow control instructions fall into two sub-groups, those that compare two values and set the condition codes, and those that which change the flow of execution (branch).

4.3.1 Comparisons

There are four comparison instructions which use the same format as the arithmetic instructions. These perform an arithmetic or logical operation on two source operands, one of which must be a register, and the other can be any $\langle op1 \rangle$ value. The instructions do not write the result to a register, but use it to update the condition flags.

The `CMP` (Compare) instructions compares two numbers. It will subtract the second operand (the $\langle op1 \rangle$ value) from the first (the register), setting all of the status flags accordingly:

Zero	Set if the two number are equal, otherwise it is clear.
Negative	Set should the second operand be larger than the first, and clear if it is smaller.
Carry	Set if the subtraction had to borrow, (unsigned underflow)
oVerflow	Set if there was a signed overflow

This will allow the full set of condition codes, general, signed, and unsigned, to be used in the instructions which follow.

The `CMN` (Compare Negative) instruction compares the register value with the negative of the second value. Thus it can be used to change the sign of the second value. Although this may cause difficulties if the value is zero.

The `TEQ` (Test Equivalence) instruction checks to see if the two values are equal. It will set the Z flag if they are, so the `EQ` (Equal) and `NE` (Not Equal) condition codes can be used. It can also be used to check if two values have the same sign. The N flag will be clear should the two values have the same sign, so the `PL` (Plus) condition can be used. Should the two values have different signs the flag will be clear and the `MI` (Minus) condition code could be used.

The final instruction in this sub-group is the `TST` (Test) instruction, which is used to discover whether a single bit of the register is set or not. The Z flag will be set if the bit in the register is clear and the `EQ` (Equal) condition code can be used. If the bit in the register was set, the flag will be clear and the `NE` (Not Equal) condition code can be used.

The `TST` can be used to test a collection of bits. However, *all* of the bits must be clear for the Z flag to be set. If *any* of the register bits are set, the flag will be cleared.

4.3.2 Branching

The `B`⟨*cc*⟩ (Branch) instruction is used to change the control flow by adding an offset to the current *Program Counter*. The offset is a signed number, allowing forward and backward branches of up to 32MB. In assembler we place a label after the `B`⟨*cc*⟩ (Branch) instruction, and the assembler will calculate the offset for us.

The Branch instruction can be used in two ways, conditionally and unconditionally. An unconditional branch will always branch no matter what the state of the flags, conventionally this is written as `BAL` (Branch Always). For a conditional branch, one of the condition codes, listed in table 4.3 on page 42, is given immediately after the `B`. Thus the `BEQ` instruction is used to branch on equal condition (the Z flag is set).

There is also the `BL`⟨*cc*⟩ (Branch and Link) instruction which preserves the address of the instruction immediately after the branch in the Link Register (LR or R*14*). This allows for a subroutine call. The code which has been called (the target of the branch) can return to the next instruction, after the branch by copying the Link Register (LR) into the Program Counter (PC). Chapters **??** and 14 go into detail over the use of subroutines.

4.3.3 Jumping

All of the instructions in the Arithmetic, Logical, and Data Movement groups can use the *Program Counter* (PC or R*15*) as a destination register. This allows them to alter the location from which the next instruction is fetched. The direct manipulation of the PC in this way is known as a *jump*. Any 32-bit value can be copied into the PC, thus a jump can be to any location in the 4GB memory space.

This is a particularly useful feature when the location of the next instruction can be calculated, via a jump table (see 13.1.5 on page 121 for further discussion).

4.4 Memory Access

The memory access instructions naturally fall into three sub-groups: Register; Register Byte; and Multiple Registers. Each of which has a Load and a Store instruction.

4.4.1 Load and Store Register

The `LDR` (Load Register) instruction can load a 32-bit word from memory into a register. While, surprisingly, the `STR` (Store Register) instructions will store a 32-bit word from a register to memory.

The Load and Store Register instructions use an ⟨*op2*⟩ value to identify the memory to use. All of the ⟨*op2*⟩ values use a *base register* and an *offset* of some form specified by the instruction:

Offset addressing is where the memory address is formed by adding (or subtracting) an offset to (or from) the value held in the base register.

Pre-indexed addressing is where the memory address is formed in the same way as for offset addressing. As a added extra the memory address is also written back into the base register. This is quite useful for looping though data, see Chapter 9 for more details.

Post-indexed addressing is where the memory address is taken from the base register directly, without any modification. This time the base register is modified (the offset is added or subtracted) after the load or store has occurred.

See section 5.2 for a full discussion of the values available for an ⟨*op2*⟩ value.

4.4.2 Load and Store Register Byte

Rather like the Load and Store Register instruction the `LDRB` (Load Register Byte) and `STRB` (Store Register Byte) instructions can load an 8-bit byte from memory, or store one respectively. The memory location is identified by an ⟨*op2*⟩ value in the same way.

The load byte instruction, will load the byte into the lower 8-bits of the register. The upper 24-bit of the register are cleared, set to zero. The byte can not be signed unless additional steps are taken.

When storing a byte, only the lower 8-bits of the register are stored. The remaining 24-bits are ignored.

4.4.3 Load and Store Multiple registers

The `LDM` (Load Multiple) and `STM` (Store Multiple) instructions allow a block transfer of any number of registers to or from memory. The memory location must be in a base register, which can be optionally updated after the transfer.

A list of registers to transfer is given, this can include the Program Counter. Each register is transferred to or from memory in turn starting with the lowest register (`R0` if given) and ending with the highest register (`R15` if given in the register list). The base register is modified with transfer, so the system knows the memory location for the next transfer.

The base register is modified according to one of four transfer modes:

A	B	C
0	0	0
0	1	0
1	0	0
1	1	1

C = A AND B

A	B	C
0	0	0
0	1	0
1	0	1
1	1	0

C = A BIC B

A	B	C
0	0	0
0	1	1
1	0	1
1	1	0

C = A EOR B

A	B	C
0	0	0
0	1	1
1	0	1
1	1	1

C = A ORR B

Table 4.4: True tables for Logical operations

`IB`	Increment Before
`IA`	Increment After
`DB`	Decrement Before
`DA`	Decrement After

The Load and Store Multiple instructions are particularly well suited for preserving registers during a subroutine call. A Store Multiple allows registers to be saved at the start of the subroutine, while a Load Multiple can return the registers back to their original state at the end of the routine. See Chapter 14 for a detailed discussion of subroutines.

4.5 Logical and Bit Manipulation

The five instructions in this group all preform logical operation on two (or in one case, one) source operands, one of which must be a register (R*n*) and the other is an ⟨*op1*⟩ value, producing a result which is placed in a destination register. They can also optionally update the condition code flags based on the result.

An ⟨*op1*⟩ value must pass through the Barrel Shift before it can be used. This means that any instruction which uses an ⟨*op1*⟩ value can preform an arithmetic or logical shift. This includes the `MOV` (Move) instruction on page 41. As a result of this, there is no requirement for the dedicated shift instructions found on Complex instruction Set Computers.

The `AND` (Bitwise AND) instruction will AND the bits of the first operand (the register) the those of the second operand (the ⟨*op1*⟩ value). In other words, the bit will be set in the result only if it is set in both the first and second operands, otherwise the bit will be clear. The `AND` is frequently used to mask out, or clear, bits we are not interested in.

The `BIC` (Bit Clear) instruction is intended to take over the masking function from the `AND` instruction. With this instruction a bit in the result is clear if the bit in the second operand is set, no matter what the bit is set to in the first operand. The remanding bits are left unchanged.

The `EOR` (Exclusive OR) instruction will OR the bits of the first operand with those of the second operand, except if both bits are set the resulting bit will be clear. This is known as an Exclusive OR, as it allows a bit from the first operand or the second operand, but not from both. This can be used to invert selected bits. By setting a bit in the second operand the corresponding bit in the first operand will be changed, if it was set it will become clear, if it was clear it will now be set.

The `ORR` (Bitwise OR) instruction will simply OR the bits of the first and second operands. The bit in the result will be set if the corresponding bit is set in either of the two operands. If the bit is clear in the second operand, the resulting bit will be the same as the first operand. This is known as an inclusive OR, as it include a bit set in not only the first operand, or the second operand, but also in both operands. This is frequently used to for a bit to be set.

Finally the `MVN` (Move Not) instruction is slightly different in that it only uses one operand. The result is a complete inversion of the operands value. That is each bit is switch from set to clear and

clear to set. This can be useful for making a bit-mask for use with one of the other instructions.

4.6 System Control / Privileged

4.6.1 Software Interrupt

The `SWI` (Software Interrupt) instructions cause a software Interrupt exception to occur. These are normally used to request a service from an operating system. The exception causes processor to change into a privileged mode, thus allowing a user mode task to gain access to privileged functions, but only under the supervision of an operating system.

There are three methods an operating system could use to identify which service is required.

1. A service number is provided as an immediate value associated with the `SWI`. Any additional parameters are passed between the operating system and the task though general-purpose registers. This is the most common form of communication.
2. The service number is placed in a general-purpose register. Any additional parameters are also passed in general-purpose registers.
3. The service number is placed in a data structure which follows immediately after the instruction. Additional parameters may be passed as part of the data structure, or in registers.

The precise details of the `SWI` instruction are dependent on the operating system or environment. This should be described in the developers documentation provided with the environment.

The only function used in this book is function number 17 (or `&11`). In the system we are using this represents a the `exit()` function, and is used to exit a program.

4.6.2 Semaphores

When there are two or more processes running on one or more processors they need a way of communicating in order to share common resources. This can be provided by a semaphore. Two instructions are provided that would help an Operating System to provide a semaphore.

The `SWP` (Swap) instruction reads a word value from memory (into a destination register) replacing it with another value (from a source register). The memory location is specified in yet another register. The same register can be given for the source and destination in which case the value at the memory location and the value is the register are simply exchanged.

There is also a `SWPB` (Swap Byte) instruction which preforms the same operation for a single byte rather than a full word.

4.6.3 Status Register Access

The status register can be accessed via two instructions. The `MRS` (Move to Register from Status) instruction will copy the current process status register (CPSR) into a general-purpose register. There is only one real reason for reading the status register, and that is to change one of the fields.

There is also a corresponding `MSR` (Move to Status from Register) instruction which replaces specific field(s) in the current status register. Access to the control fields (process mode and interrupt enable flags) is restricted to privileged mode. Only the condition code flags may be overwritten in User mode, this is to stop the wayward User mode program from entering into a privileged mode.

The MRS and MSR instructions may also access the saved process status register (SPSR). As the User and System modes do not have a SPSR register and the privileged modes do, these instruction are limited to privileged mode only.

The Data Movement, Arithmetic, and Logical instruction groups all allow the Program Counter as a destination register. This allows them to calculate the address of the next instruction. In addition to this, when the PC is the destination register and the $\langle S \rangle$ flag is given, these instructions will also copy the SPSR for the current mode into the CPSR. Obviously such instructions are restricted to privileged modes, as they actually have a SPSR. This is a rather peculiar special effect which has been provided to allow exception handlers to return the processor back to the same state it was in before the exception occurred.

4.6.4 Coprocessor

Up to 16 coprocessors can be added on to the processor to perform additional operations. Such coprocessors could include a Memory Management Unit (MMU), a Floating Point Unit (FPU), and a Digital Signal Processor (DSP) Unit.

Different manufactures extend or customise the ARM by the addition of various coprocessors. A discussion of such coprocessors is well beyond the scope of this book. It is however worth noting that there are five instructions for communicating with the coprocessor which fall into three sub-groups:

Data-processing instructions which start a coprocessor specific operation, CDP (Coprocessor Data Operation).

Data transfer instructions which transfer data between the coprocessor and memory. The LDC (Load Coprocessor) and STC (Store Coprocessor) instructions.

Register transfer instructions to transfer data between a register and the coprocessor. In particular the MRC (Move from Register to Coprocessor) and MCR (Move from Coprocessor to Register) instructions.

A manufacture may provide additional pseudo instructions which hide the use of the coprocessor from you.

4.6.5 Privileged Memory Access

The Memory Access instructions (4.4 on page 46) provide access to memory in the current processor mode. Thus if used when the processor is in a privileged mode, the memory access will also be privileged. The could cause difficulty if the memory being accessed is only available in User mode.

To cater for this difficulty a set of memory access instructions have been provided which access the memory in User mode, without effecting the current processor mode. These are known as the "with Translation" instructions.

LDRT (Load Register with Translation) and STRT (Store Register with Translation) provide 32-bit memory access with User mode Translation. The corresponding LDRBT (Load Register Byte with Translation) and STRBT (Store Register Byte with Translation) instructions provide the 8-bit memory access with User mode Translation.

As these instructions are used when an exception handler is required to access User-mode memory, any further discussion is beyond the scope of an introductory text.

4.6.6 Undefined Instructions

There are many instruction words (values) which have yet to be defined. There are even some which have been explicitly left undefined. Attempting to perform one of these undefined words will cause an Undefined Instruction exception to occur.

An Operating System can use this exception to provide additional services, and instructions which are specific to that operating environment. Any such additional instructions should be documented in the developers documented provided with the environment.

Should the environment not provide any additional services in this manner it should produce a suitable error response.

5 *Addressing Modes*

5.1 Data Processing Operands: ⟨*op1*⟩

The majority of the instructions relate to data processing of some form. One of the operands to these instructions is routed through the Barrel Shifter. This means that the operand can be modified before it is used. This can be very useful when dealing with lists, tables and other complex data structures. We denote instructions of this type as taking one of its arguments from ⟨*op1*⟩.

An ⟨*op1*⟩ argument may come from one of two sources, a constant value or a register, and be modified in five different ways.

5.1.1 Unmodified Value

You can use a value or a register unmodified by simply giving the value or the register name. For example the following instructions will demonstrate the two methods:

Immediate

```
MOV    R0, #1234
```

Will move the immediate constant value 1234_{10} into the register R*0*. R*0* ← IR(value)

Register

```
MOV    R0, R1
```

Will move the value in the register R*1* into the register R*0*. R*0* ← R*1*

5.1.2 Logical Shift Left

This will take the value of a register and shift the value up, towards the most significant bit, by n bits. The number of bits to shift is specified by either a constant value or another register. The lower bits of the value are replaced with a zero. This is a simple way of performing a multiply by a power of 2 ($\times 2^n$).

Logical Shift Left Immediate

```
MOV    R0, R1, LSL #2
```

R*0* will become the value of R*1* shifted left by 2 bits. The value of R*1* is not changed. $R0 \leftarrow R1 \ll \mathsf{IR(value)}$

Logical Shift Left Register

```
MOV    R0, R1, LSL R2
```

R*0* will become the value of R*1* shifted left by the number of bits specified in the R*2* register. R*0* is the only register to change, both R*1* and R*2* are not effected by this operation. $R0 \leftarrow R1 \ll R2(7{:}0)$

If the instruction is to set the status register, the carry flag (C) is the last bit that was shifted out of the value.

5.1.3 Logical Shift Right

Logical Shift Right is very similar to Logical Shift Left except it will shift the value to the right, towards the lest significant bit, by n bits. It will replace the upper bits with zeros, thus providing an efficient unsigned divide by 2^n function ($|\div 2^n|$). The number of bits to shift may be specified by either a constant value or another register.

Logical Shift Right Immediate

```
MOV    R0, R1, LSR #2
```

R*0* will take on the value of R*1* shifted to the right by 2 bits. The value of R*1* is not changed. $R0 \leftarrow R1 \gg \mathsf{IR(value)}$

Logical Shift Right Register

```
MOV    R0, R1, LSR R2
```

As before R*0* will become the value of R*1* shifted to the right by the number of bits specified in the R*2* register. R*1* and R*2* are not altered by this operation. $R0 \leftarrow R1 \gg R2$

If the instruction is to set the status register, the carry flag (C) is the last bit to be shifted out of the value.

5.1.4 Arithmetic Shift Right

The Arithmetic Shift Right is rather similar to the Logical Shift Right, but rather than replacing the upper bits with a zero, it maintains the value of the most significant bit. As the most significant bit is used to hold the sign, this means the sign of the value is maintained, thus providing a signed divide by 2^n operation ($\div 2^n$).

Arithmetic Shift Right Immediate

```
MOV    R0, R1, ASR #2
```

Register R*0* will become the value of register R*1* shifted to the right by 2 bits, with the sign maintained.

$$R0 \leftarrow R1 \twoheadrightarrow \mathsf{IR(value)}$$

Arithmetic Shift Right Register

```
MOV    R0, R1, ASR R2
```

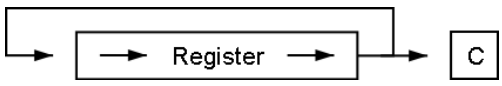
Register R*0* will become the value of the register R*1* shifted to the right by the number of bits specified by the R*2* register. R*1* and R*2* are not altered by this operation.

$$R0 \leftarrow R1 \twoheadrightarrow R2$$

Given the distinction between the Logical and Arithmetic Shift Right, why is there no Arithmetic Shift Left operation?

As a signed number is stored in two's complement the upper most bits hold the sign of the number. These bits can be considered insignificant unless the number is of a sufficient size to require their use. Thus an Arithmetic Shift Left is not required as the sign is automatically preserved by the Logical Shift.

5.1.5 Rotate Right



In the Rotate Right operation, the lest significant bit is copied into the carry (C) flag, while the value of the C flag is copied into the most significant bit of the value. In this way none of the bits in the value are lost, but are simply moved from the lower bits to the upper bits of the value.

Rotate Right Immediate

```
MOV    R0, R1, ROR #2
```

This will rotate the value of R*1* by two bits. The most significant bit of the resulting value will be the same as the least significant bit of the original value. The second most significant bit will be the same as the Carry flag. In the S version the Carry flag will be set to the second least significant bit of the original value. The value of R*1* is not changed by this operation.

$$R0 \leftarrow R1 \ggg \mathsf{IR(value)}$$

Rotate Right Register

```
MOV    R0, R1, ROR R2
```

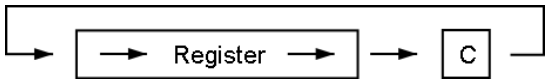
Register R*0* will become the value of the register R*1* rotated to the right by the number of bits specified by the R*2* register. R*1* and R*2* are not altered by this operation.

$$R0 \leftarrow R1 \ggg R2$$

Why is there no corresponding Rotate Left operation?

An Add With Carry (`ADC`, A.1 on page 137) to a zero value provides this service for a single bit. The designers of the instruction set believe that a Rotate Left by more than one bit would never be required, thus they have not provided a ROL function.

5.1.6 Rotate Right Extended



This is similar to a Rotate Right by one bit. The *extended* section of the fact that this function moves the value of the Carry (C) flag into the most significant bit of the value, and the least significant bit of the value into the Carry (C) flag. Thus it allows the Carry flag to be propagated though multi-word values, thereby allowing values larger than 32-bits to be used in calculations.

Rotate Right Extend

```
MOV    R0, R1 RRX
```

The register R*0* become the same as the value of the register R*1* rotated though the carry flag by one bit. The most significant bit of the value becomes the same as the current Carry flag, while the Carry flag will be the same as the least significant bit or R*1*. The value of R*1* will not be changed.

$$R0 \leftarrow C \ggg R1 \ggg C$$

5.2 Memory Access Operands: ⟨*op2*⟩

The memory address used in the memory access instructions may also modified by the barrel shifter. This provides for more advanced access to memory which is particularly useful when dealing with more advanced data structures. It allows pre- and post-increment instructions that update memory pointers as a side effect of the instruction. This makes loops which pass though memory more efficient. We denote instructions of this type as taking one of its arguments from ⟨*op2*⟩.

There are three main methods of specifying a memory address (⟨*op2*⟩), all of which include an offset value of some form. This offset can be specified in one of three ways:

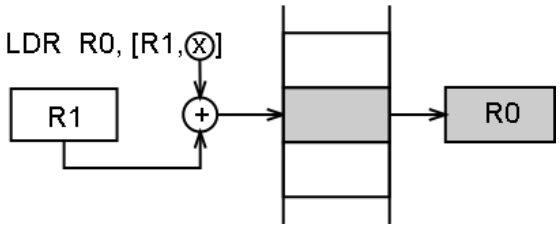
Constant Value
: An immediate constant value can be provided. If no offset is specified an immediate constant value of zero is assumed.

Register
: The offset can be specified by another register. The value of the register is added to the address held in another register to form the final address.

Scaled

The offset is specified by another register which can be scaled by one of the shift operators used for ⟨*op1*⟩. More specifically by the Logical Shift Left (LSL), Logical Shift Right (LSR), Arithmetic Shift Right (ASR), ROtate Right (ROR) or Rotate Right Extended (RRX) shift operators, where the number of bits to shift is specified as a constant value.

5.2.1 Offset Addressing



In *offset addressing* the memory address is formed by adding (or subtracting) an offset to or from the value held in a base register.

Zero Offset

```
LDR    R0, [R1]
```

Will load the register R*0* with the 32-bit word at the memory address held in the register R*1*. In this instruction there is no offset specified, so an offset of zero is assumed. The value of R*1* is not changed in this instruction.

MAR ← R*1*
MBR ← M(MAR)
R*0* ← MBR

Immediate Offset

```
LDR    R0, [R1, #4]
```

Will load the register R*0* with the word at the memory address calculated by adding the constant value 4 to the memory address contained in the R*1* register. The register R*1* is not changed by this instruction.

MAR ← R*1* + IR(value)
MBR ← M(MAR)
R*0* ← MBR

Register Offset

```
LDR    R0, [R1, R2]
```

Loads the register R*0* with the value at the memory address calculated by adding the value in the register R*1* to the value held in the register R*2*. Both R*1* and R*2* are not altered by this operation.

MAR ← R*1* + R*2*
MBR ← M(MAR)
R*0* ← MBR

Scaled Register Offset

```
LDR    R0, [R1, R2, LSL #2]
```

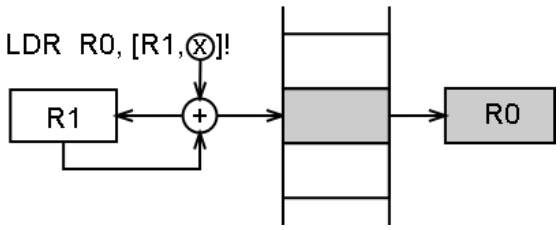
Will load the register R*0* with the 32-bit value at the memory address calculated by adding the value in the R*1* register to the value obtained by shifting the value in R*2* left by 2 bits. Both registers, R*1* and R*2* are not effected by this operation.

$$\begin{aligned} \mathsf{MAR} &\leftarrow \mathsf{R1} + \mathsf{R2} \ll \mathsf{IR(value)} \\ \mathsf{MBR} &\leftarrow \mathsf{M(MAR)} \\ \mathsf{R0} &\leftarrow \mathsf{MBR} \end{aligned}$$

This is particularly useful for indexing into a complex data structure. The start of the data structure is held in a *base* register, R*1* in this case, and the offset to access a particular field within the structure is then added to the base address. Placing the offset in a register allows it to be calculated at run time rather than fixed. This allows for looping though a table.

A scaled value can also be used to access a particular item of a table, where the size of the item is a power of two. For example, to locate item 7 in a table of 32-bit values we need only shift the index value 6 left by 2 bits (6×2^2) to calculate the value we need to add as an offset to the start of the table held in a register, R*1* in our example. Remember that the computer count from zero, thus we use an index value of 6 rather than 7. A 32-bit number requires 4 bytes of storage which is 2^2, thus we only need a 2-bit left shift.

5.2.2 Pre-Index Addressing



In *pre-index addressing* the memory address if formed in the same way as for offset addressing. The address is not only used to access memory, but the base register is also modified to hold the new value. In the ARM system this is known as a *write-back* and is denoted by placing a exclamation mark after at the end of the ⟨*op2*⟩ code.

Pre-Index address can be particularly useful in a loop as it can be used to automatically increment or decrement a counter or memory pointer.

Immediate Pre-indexed

```
LDR    R0, [R1, #4]!
```

Will load the register R*0* with the word at the memory address calculated by adding the constant value 4 to the memory address contained in the R*1* register. The new memory address is placed back into the base register, register R*1*.

$$\begin{aligned} \mathsf{R1} &\leftarrow \mathsf{R1} + \mathsf{IR(value)} \\ \mathsf{MAR} &\leftarrow \mathsf{R1} \\ \mathsf{MBR} &\leftarrow \mathsf{M(MAR)} \\ \mathsf{R0} &\leftarrow \mathsf{MBR} \end{aligned}$$

Register Pre-indexed

```
LDR    R0, [R1, R2]!
```

Loads the register R*0* with the value at the memory address calculated by adding the value in the register R*1* to the value held in the register R*2*. The offset register, R*2*, is not altered by this operation, the register holding the base address, R*1*, is modified to hold the new address.

$$
\begin{aligned}
R1 &\leftarrow R1 + R2 \\
MAR &\leftarrow R1 \\
MBR &\leftarrow M(MAR) \\
R0 &\leftarrow MBR
\end{aligned}
$$

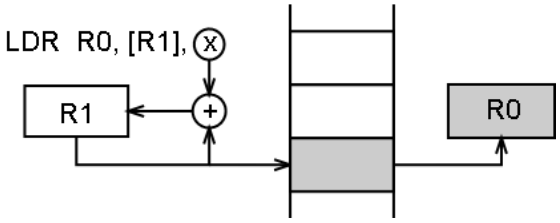
Scaled Register Pre-indexed

```
LDR    R0, [R1, R2, LSL #2]!
```

First calculates the new address by adding the value in the base address register, R*1*, to the value obtained by shifting the value in the offset register, R*2*, left by 2 bits. It will then load the 32-bit at this address into the destination register, R*0*. The new address is also written back into the base register, R*1*. The offset register, R*2*, will not be effected by this operation.

$$
\begin{aligned}
R1 &\leftarrow R1 + R2 \ll IR(value) \\
MAR &\leftarrow R1 \\
MBR &\leftarrow M(MAR) \\
R0 &\leftarrow MBR
\end{aligned}
$$

5.2.3 Post-Index Addressing



In *post-index address* the memory address is the base register value. As a side-effect, an offset is added to or subtracted from the base register value and the result is written back to the base register.

Post-index addressing uses the value of the base register without modification. It then applies the modification to the address and writes the new address back into the base register. This can be used to automatically increment or decrement a memory pointer after it has been used, so it is pointing to the next location to be used.

As the instruction must preform a write-back we do not need to include an exclamation mark. Rather we move the closing bracket to include only the base register, as that is the register holding the memory address we are going to access.

Immediate Post-indexed

```
LDR    R0, [R1], #4
```

Will load the register R*0* with the word at the memory address contained in the base register, R*1*. It will then calculate the new value of R*1* by adding the constant value 4 to the current value of R*1*.

$$
\begin{aligned}
MAR &\leftarrow R1 \\
MBR &\leftarrow M(MBR) \\
R0 &\leftarrow MBR \\
R1 &\leftarrow R1 + IR(value)
\end{aligned}
$$

Register Post-indexed

```
LDR    R0, [R1], R2
```

Loads the register R*0* with the value at the memory address held in the base register, R*1*. It will then calculate the new value for the base register by adding the value in the offset register, R*2*, to the current value of the base register. The offset register, R*2*, is not altered by this operation.

$$\begin{aligned} \text{MAR} &\leftarrow \text{R}1 \\ \text{MBR} &\leftarrow \text{M(MBR)} \\ \text{R}0 &\leftarrow \text{MBR} \\ \text{R}1 &\leftarrow \text{R}1 + \text{R}2 \end{aligned}$$

Scaled Register Post-indexed

```
LDR    R0, [R1], R2, LSL #2
```

First loads the 32-bit value at the memory address contained in the base register, R*1*, into the destination register, R*0*. It will then calculate the new value for the base register by adding the current value to the value obtained by shifting the value in the offset register, R*2*, left by 2 bits. The offset register, R*2*, will not be effected by this operation.

$$\begin{aligned} \text{MAR} &\leftarrow \text{R}1 \\ \text{MBR} &\leftarrow \text{M(MBR)} \\ \text{R}0 &\leftarrow \text{MBR} \\ \text{R}1 &\leftarrow \text{R}1 + \text{R}2 \ll \text{IR(value)} \end{aligned}$$

6 Programs

The only way to learn assembly language programming is through experience. Throughout the rest of this book each chapter will introduce various aspects of assembly programming. The chapter will start with a general discussion, then move on to a number of example programs which will demonstrate the topic under discussion. The chapter will end with a number of programming problems for you to try.

6.1 Example Programs

Each of the program examples contains several parts:

Title	that describes the general problem
Purpose	statement of purpose that describes the task the program performs and the memory locations used.
Problem	A sample problem complete with data and results.
Algorithm	if the program logic is complex.
Source code	for the assembly program.
Notes	Explanatry notes that discusses the instructions and methods used in the program.

Each example is written and assembled as a stand-alone program.

6.1.1 Program Listing Format

The examples in the book are the actual source code used to generate the programs. Sometimes you may need to use the listing output of the ARM assembler (the `.list` file), and in any case you should be aware of the fact that you can generate a listing file. See the section on the ARMulator environment which follows for details of how to generate a `.list` listing file.

6.1.2 Guidelines for Examples

We have used the following guidelines in construction of the examples:

1. Standard ARM assembler notation is used, as summarized in Chapter 2.
2. The forms in which data and addresses appear are selected for clarity rather than for consistency. We use hexadecimal numbers for memory addresses, instruction codes, and BCD data; decimal for numeric constants; binary for logical masks; and ASCII for characters.
3. Frequently used instructions and programming techniques are emphasized.
4. Examples illustrate tasks that microprocessors perform in communication, instrumentation, computers, business equipment, industrial, and military applications.

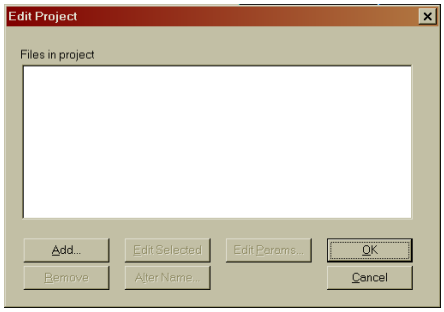


Figure 6.1: New Project Dialog

5. Detailed comments are included.
6. Simple and clear structures are emphasised, but programs are written as efficiently as possible within this guideline. Notes accompanying programs often describe more efficient procedures.
7. Program are written as an independent procedures or subroutines although no assumptions are made concerning the state of the microprocessor on entry to the procedure.
8. Program end with a `SWI &11` (Software Interrupt) instruction. You may prefer to modify this by replacing the `SWI &11` instruction with an endless loop instruction such as:

   ```
   HERE  BAL  HERE
   ```

9. Programs use standard ARM assembler directives. We introduced assembler directives conceptually in Chapter 2. When first examining programming examples, you can ignore the assembler directives if you do not understand them. Assembler directives do not contribute to program logic, which is what you will be trying to understand initially; but they are a necessary part of every assembly language program, so you will have to learn how to use them before you write any executable programs. Including assembler directives in all program examples will help you become familiar with the functions they perform.

6.2 Trying the examples

To test one of the example programs, first obtain a copy of the source code. The best way of doing this is to type in the source code presented in this book, as this will help you to understand the code. Alternatively you can download the source from the web site, although you won't gain the same knowledge of the code.

Go to the start menu and call up the "Armulate" program. Next open the source file using the normal "File | Open" menu option. This will open your program source in a separate window within the "Armulate" environment.

The next step is to create a new Project within the environment. Select the "Project" menu option, then "New". Give your project the same name as the source file that you are using (there is no need to use a file extension – it will automatically be saved as a `.apj` file).

Once you have given the file a name, a further dialog will open as shown in the figure 6.1.

Click the "Add" button, and you will again be presented with a file dialog, which will display the source files in the current directory. Select the relevant source file and "OK" the dialog. You will be returned to the previous dialog, but you will see now that your source file is included in the project. "OK" the "Edit Project" dialog, and you will be returned to the Armulate environment, now with two windows open within it, one for the source code and one for the project.

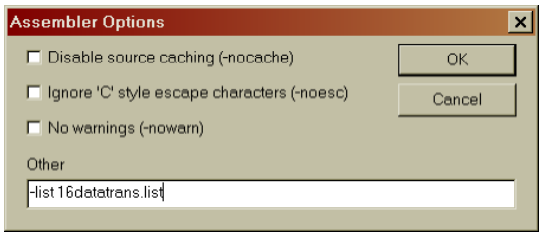


Figure 6.2: Assembler Options Dialog

We recommend that you always create a `.list` listing file for each project that you create. Do this by selecting the "Options" menu with the project window in focus, then the "Assembler" item. This will open the dialog shown in figure 6.2.

Enter `-list [yourfilename].list` into the "Other" text box and "OK" the dialog.

You have now created your project and are ready to assemble and debug your code.

Additional information on the Armulator is available via the help menu item.

6.3 Trying the examples from the command line

When developing the example programs, we found the "Armulate" environment too clumsy. We used the TextPad editor and assembled the programs from the command line. The Armulate environment provides commands for use from the command line:

1. Assembler
   The command line assembler is used to create an object file from the program source code. During the development of the add program (program 7.3a) we used the command line:

   ```
   ARMASM -LI -CPU ARM6 -g -list add.list add.s
   ```

2. Linker
   It is necessary to position the program at a fixed location in memory. This is done using the linker. In our add example we used the command:

   ```
   ARMLINK -o add add.o
   ```

   Which resolves the relative addresses in the `add.o` file, producing the `add` load image.

3. Debugger
   Finally it is necessary to debug the load image. This can be done in one of two ways, using a command line debugger or the windows debugger. In either case they require a load image (`add` in our example). To use the command line debugger (known as the source debugger) the following command is used:

   ```
   ARMSD add
   ```

   However, the command driven nature of this system is confusing and hard to use for even the most experienced of developers. Thus we suggest you use the windows based debugger program:

   ```
   WINDBG add
   ```

   Which will provide you with the same debugger you would have seen had you used the Window based Armulate environment.

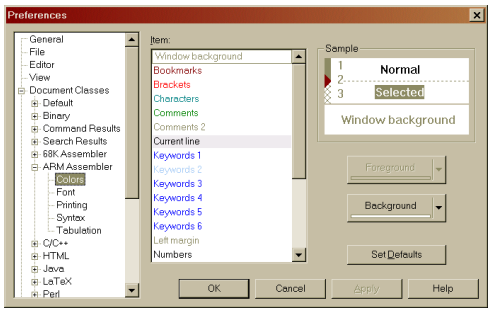


Figure 6.3: TextPad Colour Preferences Dialog

6.3.1 Setting up TextPad

To set up this environment simply download the TextPad editor and the ARM Assembler syntax file. You can download the editor from the download page of the TextPad web site[1].

Download Derek Law's ARM Assembler Syntax Definition file from the TextPad web site. You can find this under the *Syntax Definition* sub-section of the *Add-ons* section of the *Download* page. Unpack the `armasm.syn` from the `arm.zip` file into the TextPad *Samples* directory.

Having installed the Syntax Definitions you should now add a new Document Class to TextPad. Run TextPad and select the New Document Class... wizard from the Configure menu. The wizard will now take you though the following steps:

1. The Document Class requires a name. We have used the name "*ARM Assembler*".
2. The Class Members, the file name extension to associate with this document class. We associate all `.s` and `.list` files with this class: "`*.s,*.list`"
3. Syntax Highlighting. The next dialog is where we tell TextPad to use syntax highlighting, simply check the Enable Syntax Highlighting box. We now need to tell it which syntax definition file to use. If the `armasm.syn` file was placed in the *Samples* directory, this will appear in the drop down list, and should be selected.

While this will create the new document class, you will almost certainly want to change the colour settings for this document class. This class uses the different levels of Keyword colouring for different aspects of the syntax as follows:

Keywords 1	Instructions
Keywords 2	Co-processor and pseudo-instructions
Keywords 3	Shift-addresses and logical directives
Keywords 4	Registers
Keywords 5	Directives
Keywords 6	Arguments and built-in names

You will probably want to set the colour setting for all of these types to the same settings. We have set all but Keywords 2 to the same colour scheme. To alter the colour setting you should select the Preferences... option from the Configure menu.

In the "Preference" dialog (shown in figure 6.4 on the next page), open the Document Classes section and then your new document class (ARM Assembler). Now you should select the colors section. This will now allow you to change the colours for any of the given colour settings.

Finally you may like to consider adding a "File Type Filter" to the "Open File" dialog. This can be done by selecting the File Type Filter entry in the Preference dialog. Simply click on the

[1] http://www.textpad.com

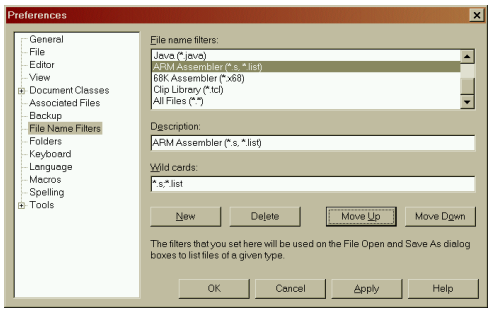


Figure 6.4: TextPad File Name Filters Preferences Dialog

New button, add the description ("ARM Assembler (*.s, *.list)") and wildcard ("*.s;*.list") details. Finally click on the OK button.

Note the use of a comma to seperate the wildcards in the description, and the use of a semi-colon (without spaces) in the wildcard entry.

6.4 Program Initialization

All of the programming examples presented in these notes pay particular attention to the correct initialization of constants and operands. Often this requires additional instructions that may appear superfluous, in that they do not contribute directly to the solution of the stated problem. Nevertheless, correct initialization is important in order to ensure the proper execution of the program every time.

We want to stress correct initialization; that is why we are going to emphasize this aspect of problems.

6.5 Special Conditions

For the same reasons that we pay particular attention to special conditions that can cause a program to fail. Empty lists and zero indexes are two of the most common circumstances overlooked in sample problems. It is critically important when using microprocessors that you learn with your very first program to anticipate unusual circumstances; they frequently cause your program to fail. You must build in the necessary programming steps to account for these potential problems.

6.6 Problems

Each chapter will now end with a number of programming problems for your to try. They have been provided to help you understand the ideas presented in the chapter. You should use the programming examples as guidelines for solving the problems. Don't forget to run your solutions on the ARMulator to ensure that they are correct.

The following guidelines will help in solving the problems:

1. Comment each program so that others can understand it. The comments can be brief and ungrammatical. They should explain the purpose of a section or instruction in the program, but should not describe the operation of instructions, that description is available in manuals. For example the following line:

```
ADD    R1, R1, #1
```

could be given the comment "Add one to R*1*" or "Increment R*1*", both of which provide no indication as to *why* the line is there. They tell us *what* the instruction is doing, but we can tell that by looking at the instruction itself. We are more interested in why the instruction is there. A comment such as "Increment loop counter" is much more useful as it explains why you are adding one to R*1*, the loop counter.

You do not have to comment each statement or explain the obvious. You may follow the format of the examples but provide less detail.

2. Emphasise clarity, simplicity, and good structure in programs. While programs should be reasonably efficient, do not worry about saving a single byte of program memory or a few microseconds.
3. Make programs reasonably general. Do not confuse parameters (such as the number of elements in any array) with fixed constants (such as the code for the letter "C").
4. Never assume fixed initial values for parameters.
5. Use assembler notation as shown in the examples and defined in Chapter 2.
6. Use symbolic notation for address and data references. Symbolic notation should also be used even for constants (such as `DATA_SELECT` instead of `2_00000100`). Also use the clearest possible form for data (such as `'C'` instead of `0x43`).
7. Use meaningful names for labels and variables, e.g., `SUM` or `CHECK` rather than `X` or `Z`.
8. Execute each program with the emulator. There is no other way of ensuring that your program is correct. We have provided sample data with each problem. Be sure that the program works for special cases.

7 Data Movement

This chapter contains some very elementary programs. They will introduce some fundamental features of the ARM. In addition, these programs demonstrate some primitive tasks that are common to assembly language programs for many different applications.

7.1 Program Examples

7.1.1 16-Bit Data Transfer

Move the contents of one 16-bit variable `Value` to another 16-bit variable `Result`.

Sample Problems

Input:	`Value`	=	C123
Output:	`Result`	=	C123

Program 7.1: `move16.s` — *16bit data transfer*

```
1   ; 16-Bit data transfer
2
3           TTL     Ch4ex1 - move16
4           AREA    Program, CODE, READONLY
5           ENTRY
6
7   Main
8           LDRB    R1, Value               ; Load the value to be moved
9           STR     R1, Result              ; Store it back in a different location
10          SWI     &11
11
12  Value   DCW     &C123                   ; Value to be moved
13          ALIGN                           ; Need to do this because working with 16bit value
14  Result  DCW     0                       ; Storage space
15
16          END
```

This program solves the problem in two simple steps. The first instruction loads data register R*1* with the 16-bit value in location `Value`. The next instruction saves the 16-bit contents of data register R*1* in location `Result`.

As a reminder of the necessary elements of an assembler program for the ARMulator, notice that this, and all the other example programs have the following elements. Firstly there must be an `ENTRY` directive. This tells the assembler where the first executable instruction is located. Next there must be at least one `AREA` directive, at the start of the program, and there may be other `AREA` directives to define data storage areas. Finally there must be an `END` directive, to show where the code ends. The absence of any of these will cause the assembly to fail with an error.

Another limitation to bear in mind is that ARMulator instructions will only deal with `BYTE` (8 bits) or `WORD` (32 bit) data sizes. It is possible to declare `HALF-WORD` (16 bit) variables by the use

of the DCW directive, but it is necessary to ensure consistency of storage of HALF-WORD by the use of the ALIGN directive. You can see the use of this in the first worked example.

In addition, under the RISC architecture of the ARM, it is not possible to directly manipulate data in storage. Even if no actual manipulation of the data is taking place, as in this first example, it is necessary to use the LDR or LDRB and STR or STRB to move data to a different area of memory.

This version of the LDR instruction moves the 32-bit word contained in memory location Value into a register and then stores it using the STR instruction at the memory location specified by Result.

Notice that, by default, every program is allocated a literal pool (a storage area) after the last executable line. In the case of this, and most of the other programs, we have formalised this by the use of the AREA Data1, DATA directive. Instruction on how to find addresses of variables will be given in the seminars.

7.1.2 One's Complement

From the bitwise complement of the contents of the 16-bit variable Value.

Sample Problems

Input:	Value	=	C123
Output:	Result	=	FFFF3EDC

Program 7.2: invert.s — *Find the one's compliment (inverse) of a number*

```
1   ; Find the one's compliment (inverse) of a number
2
3           TTL     Ch4Ex2 - invert
4           AREA    Program, CODE, READONLY
5           ENTRY
6
7   Main
8           LDR     R1, Value               ; Load the number to be complimented
9           MVN     R1, R1                  ; NOT the contents of R1
10          STR     R1, Result              ; Store the result
11          SWI     &11
12
13  Value   DCD     &C123                   ; Value to be complemented
14  Result  DCD     0                       ; Storage for result
15
16          END
```

This program solves the problem in three steps. The first instruction moves the contents of location Value into data register R*1*. The next instruction MVN takes the logical complement of data register R*1*. Finally, in the third instruction the result of the logical complement is stored in Value.

Note that any data register may be referenced in any instruction that uses data registers, but note the use of R*15* for the program counter, R*14* for the link register and R*13* for the stack pointer. Thus, in the LDR instruction we've just illustrated, any of the general purpose registers could have been used.

The LDR and STR instructions in this program, like those in Program 7.1, demonstrate one of the ARM's addressing modes. The data reference to Value as a source operand is an example of immediate addressing. In immediate addressing the offset to the address of the data being referenced (less 8 byes) is contained in the extension word(s) following the operation word of the instruction. As shown in the assembly listing, the offset to the address corresponding to Value is found in the extension word for the LDR and STR instructions.

7.1.3 32-Bit Addition

Add the contents of the 32-bit variable `Value1` to the contents of the 32-bit variable `Value2` and place the result in the 32-bit variable `Result`.

Sample Problems

Input:	`Value1`	=	37E3C123
	`Value2`	=	367402AA
Output:	`Result`	=	6E57C3CD

Program 7.3a: `add.s` — *Add two numbers*

```
1   ; Add two (32-Bit) numbers
2
3           TTL     Ch4Ex3 - add
4           AREA    Program, CODE, READONLY
5           ENTRY
6
7   Main
8           LDR     R1, Value1              ; Load the first number
9           LDR     R2, Value2              ; Load the second number
10          ADD     R1, R1, R2              ; ADD them together into R1 (x = x + y)
11          STR     R1, Result              ; Store the result
12          SWI     &11
13
14  Value1  DCD     &37E3C123               ; First value to be added
15  Value2  DCD     &367402AA               ; Second value to be added
16  Result  DCD     0                       ; Storage for result
17
18          END
```

The `ADD` instruction in this program is an example of a three-operand instruction. Unlike the `LDR` instruction, this instruction's third operand not only represents the instruction's destination but may also be used to calculate the result. The format:

DESTINATION ← SOURCE1 *operation* SOURCE2

is common to many of the instructions.

As with any microprocessor, there are many instruction sequences you can execute which will solve the same problem. Program 7.3b, for example, is a modification of Program 7.3a and uses offset addressing instead of immediate addressing.

Program 7.3b: `add2.s` — *Add two numbers and store the result*

```
1   ; Add two numbers and store the result
2
3           TTL     Ch4Ex4 - add2
4           AREA    Program, CODE, READONLY
5           ENTRY
6
7   Main
8           LDR     R0, =Value1             ; Load the address of first value
9           LDR     R1, [R0]                ; Load what is at that address
10          ADD     R0, R0, #0x4            ; Adjust the pointer
11          LDR     R2, [R0]                ; Load what is at the new addr
12          ADD     R1, R1, R2              ; ADD together
13          LDR     R0, =Result             ; Load the storage address
14          STR     R1, [R0]                ; Store the result
15          SWI     &11                     ; All done
16
17  Value1  DCD     &37E3C123               ; First value
18  Value2  DCD     &367402AA               ; Second value
19  Result  DCD     0                       ; Space to store result
```

```
20
21          END
```

The ADR pseudo-instruction introduces a new addressing mode — offest addressing, which we have not used previously. Immediate addressing lets you define a data constant and include that constant in the instruction's associated object code. The assembler format identifies immediate addressing with a # preceding the data constant. The size of the data constant varies depending on the instruction. Immediate addressing is extremely useful when small data constants must be referenced.

The ADR pseudo-instruction could be replaced by the use of the instruction LDR together with the use of the = to indicate that the address of the data should be loaded rather than the data itself.

The second addressing mode — offset addressing — uses immediate addressing to load a pointer to a memory address into one of the general purpose registers.

Program 7.3b also demonstrates the use of base register plus offset addressing. In this example we have performed this operation manually on line 10 (ADD R0, R0, #0x4), which increments the address stored in R0 by 4 bytes or one WORD. There are much simpler and more efficient ways of doing this, such as pre-index or post-index addressing which we will see in later examples.

Another advantage of this addressing mode is its faster execution time as compared to immediate addressing. This improvement occurs because the address extension word(s) does not have to be fetched from memory prior to the actual data reference, after the initial fetch.

A final advantage is the flexibility provided by having R0 hold an address instead of being fixed as part of the instruction. This flexibility allows the same code to be used for more than one address. Thus if you wanted to add the values contained in consecutive variables Value3 and Value4, you could simply change the contents of R0.

7.1.4 Shift Left One Bit

Shift the contents of the 16-bit variable Value to the left one bit. Store the result back in Result.

Sample Problems

Input:	Value	=	4242	($0100\ 0010\ 0100\ 0010_2$)
Output:	Result	=	8484	($1000\ 0100\ 1000\ 0100_2$)

Program 7.4: shiftleft.s — *Shift Left one bit*

```
1   ; Shift Left one bit
2
3           TTL     Ch4Ex5 - shiftleft
4           AREA    Program, CODE, READONLY
5           ENTRY
6
7   Main
8           LDR     R1, Value               ; Load the value to be shifted
9           MOV     R1, R1, LSL #0x1        ; SHIFT LEFT one bit
10          STR     R1, Result              ; Store the result
11          SWI     &11
12
13  Value   DCD     &4242                   ; Value to be shifted
14  Result  DCD     0                       ; Space to store result
15
16          END
```

The MOV instruction is used to perform a logical shift left. Using the operand format of the MOV instruction shown in Program 7.4, a data register can be shifted from 1 to 25 bits on either a byte,

word or longword basis. Another form of the LSL operation allows a shift counter to be specified in another data register.

7.1.5 Byte Disassembly

Divide the least significant byte of the 8-bit variable Value into two 4-bit nibbles and store one nibble in each byte of the 16-bit variable Result. The low-order four bits of the byte will be stored in the low-order four bits of the least significant byte of Result. The high-order four bits of the byte will be stored in the low-order four bits of the most significant byte of Result.

Sample Problems

Input:	Value	=	5F
Output:	Result	=	050F

Program 7.5: nibble.s — *Disassemble a byte into its high and low order nibbles*

```
1   ; Disassemble a byte into its high and low order nibbles
2
3           TTL     Ch4Ex6 - nibble
4           AREA    Program, CODE, READONLY
5           ENTRY
6
7   Main
8           LDR     R1, Value               ; Load the value to be disassembled
9           LDR     R2, Mask                ; Load the bitmask
10          MOV     R3, R1, LSR #0x4        ; Copy just the high order nibble into R3
11          MOV     R3, R3, LSL #0x8        ; Now left shift it one byte
12          AND     R1, R1, R2              ; AND the original number with the bitmask
13          ADD     R1, R1, R3              ; Add the result of that to
14                                          ; What we moved into R3
15          STR     R1, Result              ; Store the result
16          SWI     &11
17
18  Value   DCB     &5F                     ; Value to be shifted
19          ALIGN                           ; Keep the memory boundaries
20  Mask    DCW     &000F                   ; Bitmask = %0000000000001111
21          ALIGN
22  Result  DCD     0                       ; Space to store result
23
24          END
```

This is an example of byte manipulation. The ARM allows most instructions which operate on words also to operate on bytes. Thus, by using the B suffix, all the LDRinstructions in Program 7.5 become LDRB instructions, therefore performing byte operations. The STR instruction must remain, since we are storing a *halfword* value. If we were only dealing with a one byte result, we could use the STRB byte version of the store instruction.

Remember that the MOV instruction performs register-to-register transfers. This use of the MOV instruction is quite frequent.

Generally, it is more efficient in terms of program memory usage and execution time to minimise references to memory.

7.1.6 Find Larger of Two Numbers

Find the larger of two 32-bit variables Value1 and Value2. Place the result in the variable Result. Assume the values are unsigned.

Sample Problems

Compare Condition	Signed	Unsigned
greater than or equal	BGE	BHS
greater than	BGT	BHI
equal	BEQ	BEQ
not equal	BNE	BNE
less than or equal	BLE	BLS
less than	BLT	BLO

Table 7.1: Signed/Unsigned Comparisons

			a	b
Input:	Value1	=	12345678	12345678
	Value2	=	87654321	0ABCDEF1
Output:	Result	=	87654321	12345678

Program 7.6: `bigger.s` — *Find the larger of two numbers*

```
1   ; Find the larger of two numbers
2
3           TTL     Ch4Ex7 - bigger
4           AREA    Program, CODE, READONLY
5           ENTRY
6
7   Main
8           LDR     R1, Value1              ; Load the first value to be compared
9           LDR     R2, Value2              ; Load the second value to be compared
10          CMP     R1, R2                  ; Compare them
11          BHI     Done                    ; If R1 contains the highest
12          MOV     R1, R2                  ; otherwise overwrite R1
13  Done
14          STR     R1, Result              ; Store the result
15          SWI     &11
16
17  Value1  DCD     &12345678               ; Value to be compared
18  Value2  DCD     &87654321               ; Value to be compared
19  Result  DCD     0                       ; Space to store result
20
21          END
```

The Compare instruction, `CMP`, sets the status register flags as if the destination, R*1*, were subtracted from the source R*2*. The order of the operands is the same as the operands in the subtract instruction, `SUB`.

The conditional transfer instruction `BHI` transfers control to the statement labeled `Done` if the unsigned contents of R*2* are greater than or equal to the contents of R*1*. Otherwise, the next instruction (on line 12) is executed. At `Done`, register R*2* will always contain the larger of the two values.

The `BHI` instruction is one of several conditional branch instructions. To change the program to operate on signed numbers, simply change the `BHI` to `BGE` (Branch if Greater than or Equal to):

```
...
CMP   R1, R2
BGE   Done
...
```

You can use the following table 8.1 to use when performing signed and unsigned comparisons.

Note that the same instructions are used for signal and unsigned addition, subtraction, or comparison; however, the comparison operations are different.

The conditional branch instructions are an example of program counter relative addressing. In other words, if the branch condition is satisfied, control will be transfered to an address relative to the current value of the program counter. Dealing with compares and branches is an important part of programming. Don't confuse the sense of the `CMP` instruction. After a compare, the relation tested is:

DESTINATION *condition* SOURCE

For exampe, if the condition is "less than," then you test for destination less than source. Become familiar with all of the conditions and their meanings. Unsigned compares are very useful when comparing two addresses.

7.1.7 64-Bit Adition

Add the contents of two 64-bit variables `Value1` and `Value2`. Store the result in `Result`.

Sample Problems

Input:	`Value1`	=	12A2E640,	F2100123
	`Value2`	=	001019BF,	40023F51
Output:	`Result`	=	12B30000,	32124074

Program 7.7: `add64.s` — *64 bit addition*

```
1   ; 64 bit addition
2
3           TTL     Ch4Ex8 - add64
4           AREA    Program, CODE, READONLY
5           ENTRY
6
7   Main
8           LDR     R0, =Value1             ; Pointer to first value
9           LDR     R1, [R0]                ; Load first part of value1
10          LDR     R2, [R0, #4]            ; Load lower part of value1
11          LDR     R0, =Value2             ; Pointer to second value
12          LDR     R3, [R0]                ; Load upper part of value2
13          LDR     R4, [R0, #4]            ; Load lower part of value2
14          ADDS    R6, R2, R4              ; Add lower 4 bytes and set carry flag
15          ADC     R5, R1, R3              ; Add upper 4 bytes including carry
16          LDR     R0, =Result             ; Pointer to Result
17          STR     R5, [R0]                ; Store upper part of result
18
19          STR     R6, [R0, #4]            ; Store lower part of result
20          SWI     &11
21
22  Value1  DCD     &12A2E640, &F2100123    ; Value to be added
23  Value2  DCD     &001019BF, &40023F51    ; Value to be added
24  Result  DCD     0                       ; Space to store result
25
26          END
```

Here we introduce several important and powerful instructions from the ARM instruction set. As before, at line 8 we use the `LDR` instruction which causes register R*0* to hold the starting address of `Value1`. At line 9 the instruction `LDR  R1, [R0]` fetches the first 4 bytes (32-bits) of the 64-bit value, starting at the location pointed to by R*0* and places them in the R*1* register. Line 10 loads the second 4 bytes or the lower half of the 64-bit value from the memroy address pointed to by R*0* plus 4 bytes (`[R0, #4]`. Between them R*1* and R*2* now hold the first 64-bit value, R*1* has the upper half while R*2* has the lower half. Lines 11–13 repeat this process for the second 64-bit value, reading it into R*3* and R*4*.

Next, the two low order *word*s, held in R*2* and R*4* are added, and the result stored in R*6*.

This is all straightforward, but note now the use of the `S` suffix to the `ADD` instruction. This forces the update of the flags as a result of the `ADD` operation. In other words, if the result of the addition results in a carry, the carry flag bit will be set.

Now the `ADC` (add with carry) instruction is used to add the two high order *word*s, held in `R1` and `R3`, but taking into account any carry resulting from the previous addition.

Finally, the result is stored using the same technique as we used the load the values (lines 16–18).

7.1.8 Table of Factorials

Calculate the factorial of the 8-bit variable `Value` from a table of factorials `DataTable`. Store the result in the 16-bit variable `Result`. Assume `Value` has a value between 0 and 7.

Sample Problems

Input:	`FTABLE`	=	0001	(0!	=	1_{10})
		=	0001	(1!	=	1_{10})
		=	0002	(2!	=	2_{10})
		=	0006	(3!	=	6_{10})
		=	0018	(4!	=	24_{10})
		=	0078	(5!	=	120_{10})
		=	02D0	(6!	=	720_{10})
		=	13B0	(7!	=	5040_{10})
	`Value`	=	05			
Output:	`Result`	=	0078	(5!	=	120_{10})

Program 7.8: `factorial.s` — *Lookup the factorial from a table by using the address of the memory location*

```
1   ; Lookup the factorial from a table using the address of the memory location
2
3           TTL     Ch4Ex9 - factorial
4           AREA    Program, CODE, READONLY
5           ENTRY
6
7   Main
8           LDR     R0, =DataTable          ; Load the address of the lookup table
9           LDR     R1, Value               ; Offset of value to be looked up
10          MOV     R1, R1, LSL #0x2        ; Data is declared as 32bit - need
11                                          ; to quadruple the offset to point at the
12                                          ; correct memory location
13          ADD     R0, R0, R1              ; R0 now contains memory address to store
14          LDR     R2, [R0]
15          LDR     R3, =Result             ; The address where we want to store the answer
16          STR     R2, [R3]                ; Store the answer
17
18          SWI     &11
19
20          AREA    DataTable, DATA
21
22          DCD     1               ;0! = 1         ; The data table containing the factorials
23          DCD     1               ;1! = 1
24          DCD     2               ;2! = 2
25          DCD     6               ;3! = 6
26          DCD     24              ;4! = 24
27          DCD     120             ;5! = 120
28          DCD     720             ;6! = 720
29          DCD     5040            ;7! = 5040
30  Value   DCB     5
31          ALIGN
32  Result  DCW     0
33
34          END
```

The approach to this table lookup problem, as implemented in this program, demonstrates the use of offset addressing. The first two LDR instructions, load register R0 with the start address of the lookup table[1], and register R1 contents of Value.

The actual calculation of the entry in the table is determined by the first operand of the R1, R1, LSL #0x2 instruction. The long word contents of address register R1 are added to the long word contents of data register R0 to form the effective address used to index the table entry. When R0 is used in this manner, it is referred to as an index register.

7.2 Problems

7.2.1 64-Bit Data Transfer

Move the contents of the 64-bit variable VALUE to the 64-bit variable RESULT.

Sample Problems

Input:	VALUE	3E2A42A1 21F260A0
Output:	RESULT	3E2A42A1 21F260A0

7.2.2 32-Bit Subtraction

Subtract the contents of the 32-bit variable VALUE1 from the contents of the 32-bit variable VALUE2 and store the result back in VALUE1.

Sample Problems

Input:	VALUE1	12343977
	VALUE2	56782182
Output:	VALUE1	4443E80B

7.2.3 Shift Right Three Bits

Shift the contents of the 32-bit variable VALUE right three bits. Clear the three most significant bit postition.

Sample Problems

		Test A	Test B
Input:	VALUE	415D7834	9284C15D
Output:	VALUE	082BAF06	1250982B

7.2.4 Halfword Assembly

Combine the low four bits of each of the four consecutive bytes beginning at LIST into one 16-bit halfword. The value at LIST goes into the most significant nibble of the result. Store the result in the 32-bit variable RESULT.

[1] Note that we are using a LDR instruction as the data table is sufficently far away from the instruction that an ADR instruction is not valid.

Sample Problems

Input:	`LIST`	0C
		02
		06
		09
Output:	`RESULT`	0000C269

7.2.5 Find Smallest of Three Numbers

The three 32-bit variables `VALUE1`, `VALUE2` and `VALUE3`, each contain an unsigned number. Store the smallest of these numbers in the 32-bit variable `RESULT`.

Sample Problems

Input:	`VALUE1`	91258465
	`VALUE2`	102C2056
	`VALUE3`	70409254
Output:	`RESULT`	102C2056

7.2.6 Sum of Squares

Calculate the squares of the contents of word `VALUE1` and word `VALUE2` then add them together. Please the result into the word `RESULT`.

Sample Problems

Input:	`VALUE1`	00000007
	`VALUE2`	00000032
Output:	`RESULT`	000009F5

That is $7^2 + 50^2 = 49 + 2500 = 2549$ (decimal)
or $7^2 + 32^2 = 31 + 9C4 = 9F5$ (hexadecimal)

7.2.7 Shift Left n bits

Shift the contents of the word `VALUE` left. The number of bits to shift is contained in the word `COUNT`. Assume that the shift count is less than 32. The low-order bits should be cleared.

Sample Problems

		Test A	Test B
Input:	`VALUE`	182B	182B
	`COUNT`	0003	0020
Output:	`VALUE`	C158	0000

In the first case the value is to be shifted left by three bits, while in the second case the same value is to be shifted by thirty two bits.

8 Logic

8.1 Program Examples

8.1.1 Find Larger of Two Numbers

Find the larger of two 32-bit variables `Value1` and `Value2`. Place the result in the variable `Result`. Assume the values are unsigned.

Sample Problems

			a	b
Input:	`Value1`	=	12345678	12345678
	`Value2`	=	87654321	0ABCDEF1
Output:	`Result`	=	87654321	12345678

Program 8.1: `bigger.s` — *Find the larger of two numbers*

```
1   ; Find the larger of two numbers
2
3           TTL     Ch4Ex7 - bigger
4           AREA    Program, CODE, READONLY
5           ENTRY
6
7   Main
8           LDR     R1, Value1              ; Load the first value to be compared
9           LDR     R2, Value2              ; Load the second value to be compared
10          CMP     R1, R2                  ; Compare them
11          BHI     Done                    ; If R1 contains the highest
12          MOV     R1, R2                  ; otherwise overwrite R1
13  Done
14          STR     R1, Result              ; Store the result
15          SWI     &11
16
17  Value1  DCD     &12345678               ; Value to be compared
18  Value2  DCD     &87654321               ; Value to be compared
19  Result  DCD     0                       ; Space to store result
20
21          END
```

The Compare instruction, `CMP`, sets the status register flags as if the destination, R*1*, were subtracted from the source R*2*. The order of the operands is the same as the operands in the subtract instruction, `SUB`.

The conditional transfer instruction `BHI` transfers control to the statement labelled `Done` if the unsigned contents of R*2* are greater than or equal to the contents of R*1*. Otherwise, the next instruction (on line 12) is executed. At `Done`, register R*2* will always contain the larger of the two values.

Compare Condition	Signed	Unsigned
greater than or equal	BGE	BCC
greater than	BGT	BHI
equal	BEQ	BEQ
not equal	BNE	BNE
less than or equal	BLS	BLS
less than	BLT	BCS

Table 8.1: Signed/Unsigned Comparisons

The BHI instruction is one of several conditional branch instructions. To change the program to operate on signed numbers, simply change the BHI to BGE (Branch if Greater than or Equal to):

```
...
CMP   R1,R2
BGE   Done
...
```

Table 8.1 gives the branch instructions to use when performing signed and unsigned comparisons.

Note that the same instructions are used for signed and unsigned addition, subtraction, or comparison; however, the comparison operations are different.

The conditional branch instructions are an example of program counter relative addressing. In other words, if the branch condition is satisfied, control will be transferred to an address relative to the current value of the program counter.

Dealing with compares and branches is an important part of programming. Don't confuse the sense of the CMP instruction. After a compare, the relation tested is:

DESTINATION *condition* SOURCE

For example, if the condition is "less than," then you test for destination less than source. Become familiar with all of the conditions and their meanings. Unsigned compares are very useful when comparing two addresses.

8.1.2 64-Bit Adition

Add the contents of two 64-bit variables Value1 and Value2. Store the result in Result.

Sample Problems

Input:	Value1	=	12A2E640,	F2100123
	Value2	=	001019BF,	40023F51
Output:	Result	=	12B30000,	32124074

Program 8.2: add64.s — *64 bit addition*

```
1   ; 64 bit addition
2
3           TTL     Ch4Ex8 - add64
4           AREA    Program, CODE, READONLY
5           ENTRY
6
7   Main
8           LDR     R0, =Value1             ; Pointer to first value
9           LDR     R1, [R0]                ; Load first part of value1
10          LDR     R2, [R0, #4]            ; Load lower part of value1
11          LDR     R0, =Value2             ; Pointer to second value
12          LDR     R3, [R0]                ; Load upper part of value2
```

```
13          LDR     R4, [R0, #4]            ; Load lower part of value2
14          ADDS    R6, R2, R4              ; Add lower 4 bytes and set carry flag
15          ADC     R5, R1, R3              ; Add upper 4 bytes including carry
16          LDR     R0, =Result             ; Pointer to Result
17          STR     R5, [R0]                ; Store upper part of result
18
19          STR     R6, [R0, #4]            ; Store lower part of result
20          SWI     &11
21
22  Value1  DCD     &12A2E640, &F2100123    ; Value to be added
23  Value2  DCD     &001019BF, &40023F51    ; Value to be added
24  Result  DCD     0                       ; Space to store result
25
26          END
```

Here we introduce several important and powerful instructions from the ARM instruction set. The first of these is `LDMIA`, which may appear to be a confusing acronym, but which simply stands for "Load many increment after". As before, at line 8 we use the `ADR` instruction which causes register `R0` to hold the starting address of `Value1`. At line 9 the instruction `LDMIA R0, {R1,R2}` fetches the next two 32-bit values, starting at the location pointed to by R*0* and places them in registers R*1* and R*2*. This process is repeated for the next two 32-bit values.

Next, the two low order `WORD`s, held in R*2* and R*4* are added, and the result stored in R*6*.

This is all straightforward, but note now the use of the `S` suffix to the `ADD` instruction. This forces the update of the flags as a result of the `ADD` operation. In other words, if the result of the addition results in a carry, the carry flag bit will be set.

Now the `ADC` (add with carry) instruction is used to add the two high order `WORD`s, held in R*1* and R*3*, but taking into account any carry resulting from the previous addition.

Finally, the `STMIA` instruction is used to store the result now held in R*5* and R*6*.

8.1.3 Table of Factorials

Calculate the factorial of the 8-bit variable `Value` from a table of factorials `DataTable`. Store the result in the 16-bit variable `Result`. Assume `Value` has a value between 0 and 7.

Sample Problems

Input:	`FTABLE`	=	0001	(0!	=	1_{10})
		=	0001	(1!	=	1_{10})
		=	0002	(2!	=	2_{10})
		=	0006	(3!	=	6_{10})
		=	0018	(4!	=	24_{10})
		=	0078	(5!	=	120_{10})
		=	02D0	(6!	=	720_{10})
		=	13B0	(7!	=	5040_{10})
	`Value`	=	05			
Output:	`Result`	=	0078	(5!	=	120_{10})

Program 8.3: `factorial.s` — *Lookup the factorial from a table by using the address of the memory location*

```
1   ; Lookup the factorial from a table using the address of the memory location
2
3           TTL     Ch4Ex9 - factorial
4           AREA    Program, CODE, READONLY
5           ENTRY
6
7   Main
8           LDR     R0, =DataTable          ; Load the address of the lookup table
```

```
 9          LDR     R1, Value               ; Offset of value to be looked up
10          MOV     R1, R1, LSL #0x2        ; Data is declared as 32bit - need
11                                          ; to quadruple the offset to point at the
12                                          ; correct memory location
13          ADD     R0, R0, R1              ; R0 now contains memory address to store
14          LDR     R2, [R0]
15          LDR     R3, =Result             ; The address where we want to store the answer
16          STR     R2, [R3]                ; Store the answer
17
18          SWI     &11
19
20          AREA    DataTable, DATA
21
22          DCD     1               ;0! = 1         ; The data table containing the factorials
23          DCD     1               ;1! = 1
24          DCD     2               ;2! = 2
25          DCD     6               ;3! = 6
26          DCD     24              ;4! = 24
27          DCD     120             ;5! = 120
28          DCD     720             ;6! = 720
29          DCD     5040            ;7! = 5040
30  Value   DCB     5
31          ALIGN
32  Result  DCW     0
33
34          END
```

The approach to this table lookup problem, as implemented in this program, demonstrates the use of offset addressing. The first two `LDR` instructions, load register `R0` with the start address of the lookup table[1], and register `R1` contents of `Value`.

The actual calculation of the entry in the table is determined by the first operand of the `R1, R1, LSL #0x2` instruction. The long word contents of address register `R1` are added to the long word contents of data register `R0` to form the effective address used to index the table entry. When `R0` is used in this manner, it is referred to as an index register.

8.2 Problems

8.2.1 64-Bit Data Transfer

Move the contents of the 64-bit variable `VALUE` to the 64-bit variable `RESULT`.

Sample Problems

Input:	`VALUE`	3E2A42A1 21F260A0
Output:	`RESULT`	3E2A42A1 21F260A0

8.2.2 32-Bit Subtraction

Subtract the contents of the 32-bit variable `VALUE1` from the contents of the 32-bit variable `VALUE2` and store the result back in `VALUE1`.

Sample Problems

[1] Note that we are using a `LDR` instruction as the data table is sufficently far away from the instruction that an `ADR` instruction is not valid.

Input:	VALUE1	12343977
	VALUE2	56782182
Output:	VALUE1	68AC5AF9

8.2.3 Shift Right Three Bits

Shift the contents of the 32-bit variable VALUE right three bits. Clear the three most significant bit position.

Sample Problems

		Test A	Test B
Input:	VALUE	415D7834	9284C15D
Output:	VALUE	082BAF06	1250982B

8.2.4 Halfword Assembly

Combine the low four bits of each of the four consecutive bytes beginning at LIST into one 16-bit halfword. The value at LIST goes into the most significant nibble of the result. Store the result in the 32-bit variable RESULT.

Sample Problems

Input:	LIST	0C
		02
		06
		09
Output:	RESULT	0000C269

8.2.5 Find Smallest of Three Numbers

The three 32-bit variables VALUE1, VALUE2 and VALUE3, each contain an unsigned number. Store the smallest of these numbers in the 32-bit variable RESULT.

Sample Problems

Input:	VALUE1	91258465
	VALUE2	102C2056
	VALUE3	70409254
Output:	RESULT	102C2056

8.2.6 Sum of Squares

Calculate the squares of the contents of word VALUE1 and word VALUE2 then add them together. Please the result into the word RESULT.

Sample Problems

Input:	VALUE1	00000007
	VALUE2	00000032
Output:	RESULT	000009F5

That is $7^2 + 50^2 = 49 + 2500 = 2549$ (decimal)
or $7^2 + 32^2 = 31 + 9C4 = 9F5$ (hexadecimal)

8.2.7 Shift Left n bits

Shift the contents of the word `VALUE` left. The number of bits to shift is contained in the word `COUNT`. Assume that the shift count is less than 32. The low-order bits should be cleared.

Sample Problems

		Test A	Test B
Input:	`VALUE`	182B	182B
	`COUNT`	0003	0020
Output:	`VALUE`	C158	0000

In the first case the value is to be shifted left by three bits, while in the second case the same value is to be shifted by thirty two bits.

9 *Program Loops*

The program loop is the basic structure that forces the CPU to repeat a sequence of instructions. Loops have four sections:

1. The initialisation section, which establishes the starting values of counters, pointers, and other variables.
2. The processing section, where the actual data manipulation occurs. This is the section that does the work.
3. The loop control section, which updates counters and pointers for the next iteration.
4. The concluding section, that may be needed to analyse and store the results.

The computer performs Sections 1 and 4 only once, while it may perform Sections 2 and 3 many times. Therefore, the execution time of the loop depends mainly on the execution time of Sections 2 and 3. Those sections should execute as quickly as possible, while the execution times of Sections 1 and 4 have less effect on overall program speed.

There are typically two methods of programming a loop, these are the "repeat ... until" loop (Algorithm 9.1a) and the "while" loop (Algorithm 9.1b). The repeat-until loop results in the computer always executing the processing section of the loop at least once. On the other hand, the computer may not execute the processing section of the while loop at all. The repeat-until loop is more natural, but the while loop is often more efficient and eliminates the problem of going through the processing sequence once even where there is no data for it to handle.

Algorithm 9.1a

```
Initialisation Section
Repeat
   Processing Section
   Loop Control Section
Until task completed
Concluding Section
```

Algorithm 9.1b

```
Initialisation Section
While task incomplete
   Processing Section
Repeat
```

The computer can use the loop structure to process large sets of data (usually called "arrays"). The simplest way to use one sequence of instructions to handle an array of data is to have the program increment a register (usually an index register or stack pointer) after each iteration. Then the register will contain the address of the next element in the array when the computer repeats the sequence of instructions. The computer can then handle arrays of any length with a single program.

Register indirect addressing is the key to the processing arrays since it allows you to vary the actual address of the data (the "*effective address*") by changing the contents of a register. The autoincrementing mode is particularly convenient for processing arrays since it automatically updates the register for the next iteration. No additional instruction is necessary. You can even have an automatic increment by 2 or 4 if the array contains 16-bit or 32-bit data or addresses.

Although our examples show the processing of arrays with autoincrementing (adding 1, 2, or 4 after each iteration), the procedure is equally valid with autodecrementing (subtracting 1, 2, or 4 before each iteration). Many programmers find moving backward through an array somewhat awkward and difficult to follow, but it is more efficient in many situations. The computer obviously does not know backward from forward. The programmer, however, must remember that the processor

increments an address register after using it but decrements an address register before using it. This difference affects initialisation as follows:

1. When moving forward through an array (autoincrementing), start the register pointing to the lowest address occupied by the array.

2. When moving backward through an array (autodecrementing), start the register pointing one step (1, 2, or 4) beyond the highest address occupied by the array.

9.1 Program Examples

9.1.1 Sum of Numbers

Algorithm 9.1a

8	Pointer ← `Table`	R0
9	Sum ← 0	R1
10	Count ← M(`Length`)	
11	Repeat	
12	temp ← M(Pointer)	R2
13	Sum ← Sum + temp	
14	Pointer ← Pointer + 4	
15	Count ← Count − 1	
15–16	Until Count = 0	
17	M(`Result`) ← Sum	

Program 9.1a: `sum1.s` — *Add a table of 32 bit numbers*

```
1   *       Add a series of 16 bit numbers by using a table address look-up
2
3           TTL     Ch5Ex1
4           AREA    Program, CODE, READONLY
5           ENTRY
6
7   Main
8           LDR     R0, =Table              ; load the address of the lookup table
9           EOR     R1, R1, R1              ; clear R1 to store sum
10          LDR     R2, Length              ; init element count
11  Loop
12          LDR     R3, [R0]                ; get the data
13          ADD     R1, R1, R3              ; add it to r1
14          ADD     R0, R0, #+4             ; increment pointer
15          SUBS    R2, R2, #0x1            ; decrement count with zero set
16          BNE     Loop                    ; if zero flag is not set, loop
17          STR     R1, Result              ; otherwise done - store result
18          SWI     &11
19
20          AREA    Data1, DATA
21
22  Table   DCW     &2040                   ; table of values to be added
23          DCW     &1C22
24          DCW     &0242
25  TablEnd DCD     0
26
27          AREA    Data2, DATA
28  Length  DCW     (TablEnd - Table) / 4   ; gives the loop count
29  Result  DCW     0                       ; storage for result
30
31          END
```

Algorithm 9.1b

8	Pointer ← `Table`	R0
9	Sum ← 0	R1
10	Count ← M(`Length`)	R2
11–12	If Count ≠ 0 Then	
15	Repeat	
16	temp ← M(Pointer)	R3
17	Sum ← Sum + temp	
18	Pointer ← Pointer + 4	
19	Count ← Count − 1	
19–20	Until Count = 0	
22	End If	
23	M(`Result`) ← Sum	

Program 9.1b: `sum2.s` — *Add a table of 32 bit numbers*

```
1   *       Add a table of 32 bit numbers
2   *       This example has an empty table, and the program handles this
3
4           TTL     Ch5Ex2
5           AREA    Program, CODE, READONLY
6           ENTRY
7
8   Main
9           LDR     R0, =Table              ; load the address of the lookup table
10          EOR     R1, R1, R1              ; clear R1 to store sum
11          LDR     R2, Length              ; init element count
12          CMP     R2, #0                  ; is count zero?
13          BEQ     Done                    ; Yes => Then we are Done
14
15  Loop
16          LDR     R3, [R0]                ; get the data that R0 points to
17          ADD     R1, R1, R3              ; add it to r1
18          ADD     R0, R0, #+4             ; increment pointer
19          SUBS    R2, R2, #0x1            ; decrement count with zero set
20          BNE     Loop                    ; if zero flag is not set, loop
21
22  Done
23          STR     R1, Result              ; store result
24          SWI     &11
25
26          AREA    Data1, DATA
27
28  Table                                   ;Table is empty
29  TablEnd DCD     0
30
31          AREA    Data2, DATA
32  Length  DCW     (TablEnd - Table) / 4   ; Calculate table length
33  Result  DCW     0                       ; storage for result
34
35          END
```

9.1.2 64-bit Sum of Numbers

Program 9.2: `bigsum.s` — *Add a table of 32-bit numbers into a 64-bit number*

File *loops/bigsum.s* does not exist!

9.1.3 Number of negative elements

Algorithm 9.3

```
 8    Pointer ← Table                    R0
 9    NegCount ← 0                       R1
10    LoopCount ← M(Length)              R2
11–12 If LoopCount ≠ 0 Then
13      Repeat
14        temp ← M(Pointer)              R3
15–16     If temp ≯ 0 Then
17          NegCount ← NegCount + 1
18        End If
19        Pointer ← Pointer + 4
20        LoopCount ← LoopCount − 1
20–21   Until LoopCount = 0
22    End If
23    M(Result) ← NegCount
```

Program 9.3a: `cntneg1.s` — *Count the number of negative values in a table of 32-bit numbers*

```
1  *       Count the number of negative values in a table of 32-bit numbers
2
3          TTL     Ch5Ex3
4          AREA    Program, CODE, READONLY
5          ENTRY
6
7  Main
8          LDR     R0, =Table              ; load the address of the lookup table
9          EOR     R1, R1, R1              ; clear R1 to store count
10         LDR     R2, Length              ; init element count
11         CMP     R2, #0                  ; is table empty?
12         BEQ     Done                    ; Yes => Skip loop
13 Loop
14         LDR     R3, [R0]                ; Read data from table
15         CMP     R3, #0                  ; Is value < 0
16         BPL     Looptest                ; Yes => skip next line
17         ADD     R1, R1, #1              ; No  => increment -ve number count
18 Looptest
19         ADD     R0, R0, #+4             ; increment pointer
20         SUBS    R2, R2, #0x1            ; decrement loop count
21         BNE     Loop                    ; repeat unless count is zero
22 Done
23         STR     R1, Result              ; store result
24         SWI     &11
25
26         AREA    Data1, DATA
27
28 Table   DCD     &F1522040               ; table of values to be added
29         DCD     &7F611C22
30         DCD     &80000242
31 TablEnd DCD     0
32
33         AREA    Data2, DATA
34 Length  DCW     (TablEnd - Table) / 4   ; gives the loop count
35 Result  DCW     0                       ; storage for result
36
37         END
```

Program 9.3b: `cntneg2.s` — *Count the number of negative values in a table of 32-bit numbers*

```
1  *       Count the number of negative values in a table of 32-bit numbers
2
3          TTL     Ch5Ex4
4          AREA    Program, CODE, READONLY
5          ENTRY
6
```

```
 7  Main
 8          LDR     R0, =Table              ; load the address of the lookup table
 9          EOR     R1, R1, R1              ; clear R1 to store count
10          LDR     R2, Length              ; init element count
11          CMP     R2, #0                  ; is table empty?
12          BEQ     Done                    ; Yes => No need to scan it
13  Loop
14          LDR     R3, [R0]                ; get the data
15          MOV     R3, R3 LSL #1           ; Shift top bit into Carry
16          ADDCS   R1, R1, #1              ; Inc NegCount if Carry Set
17          ADD     R0, R0, #+4             ; Increment pointer
18          SUBS    R2, R2, #0x1            ; Decrement Loop Count
19          BNE     Loop                    ; Repeat Loop if Count not zero
20  Done
21          STR     R1, Result              ; Store result
22          SWI     &11
23
24          AREA    Data1, DATA
25
26  Table   DCD     &F1522040               ; table of values to be added
27          DCD     &7F611C22
28          DCD     &80000242
29  TablEnd DCD     0
30
31          AREA    Data2, DATA
32  Length  DCW     (TablEnd - Table) / 4   ; Calculate the loop count
33  Result  DCW     0                       ; Storage for result
34
35          END
```

9.1.4 Find Maximum Value

Algorithm 9.4

```
 8   Pointer ← Table                    R0
 9   Max ← 0                            R1
10   Count ← Length                     R2
11–12 If Count ≠ 0 Then
13     Repeat
14       temp ← M(Pointer)              R3
15–16    If temp ≰ Max Then
17         Max ← temp
18       End If
19       Pointer ← Pointer + 4
20       Count ← Count − 1
20–21  Until Count = 0
22   End If
23   M(Result) ← Max
```

Program 9.4: `largest.s` — *Find largest unsigned 32 bit value in a table*

```
 1  *       Find largest unsigned 32 bit value in a table
 2
 3          TTL     Ch5Ex5
 4          AREA    Program, CODE, READONLY
 5          ENTRY
 6
 7  Main
 8          LDR     R0, =Table              ; load the address of the lookup table
 9          EOR     R1, R1, R1              ; clear Max
10          LDR     R2, Length              ; init loop count
11          CMP     R2, #0                  ; is Loop Count empty?
12          BEQ     Done                    ; Yes => No table to search!
13  Loop
14          LDR     R3, [R0]                ; Get the data
15          CMP     R3, R1                  ; Is it Lower or Same as current Max ?
```

```
16          BLS     Looptest                ; Yes => skip next line
17          MOV     R1, R3                  ; No  => Make this the current Max
18  Looptest
19          ADD     R0, R0, #+4             ; increment pointer
20          SUBS    R2, R2, #0x1            ; decrement loop count
21          BNE     Loop                    ; Repeat if loop count not zero
22  Done
23          STR     R1, Result              ; store result
24          SWI     &11
25
26          AREA    Data1, DATA
27
28  Table   DCW     &A152                   ; table of values to be tested
29          DCW     &7F61
30          DCW     &F123
31          DCW     &8000
32  TablEnd DCD     0
33
34          AREA    Data2, DATA
35
36  Length  DCW     (TablEnd - Table) / 4   ; Calculste the loop count
37  Result  DCW     0                       ; storage for result
38
39          END
```

9.1.5 Normalise A Binary Number

Algorithm 9.5

8	Pointer ← `Table`	R0
9	Count ← 0	R1
10	Value ← M(Pointer)	R2
11–12	If Value ≠ 0 Then	
13	Repeat	
14	Count ← Count + 1	
15	Value ← Value ≪ 1	
16	Until Value[31] = 1	
17	End If	
18	M(Shift) ← Count	
19	M(Normal) ← Value	

Program 9.5a: `normal1.s` — *Normalize a binary number*

```
1   *       Normalize a Binary Number
2
3           TTL     Ch5Ex6
4           AREA    Program, CODE, READONLY
5           ENTRY
6
7   Main
8           LDR     R0, =Table              ; load the address of the lookup table
9           EOR     R1, R1, R1              ; clear R1 to store shifts
10          LDR     R3, [R0]                ; get the data
11          CMP     R3, R1                  ; bit is 1
12          BEQ     Done                    ; if table is empty
13  Loop
14          ADD     R1, R1, #1              ; increment pointer
15          MOVS    R3, R3, LSL#0x1         ; decrement count with zero set
16          BPL     Loop                    ; if negative flag is not set, loop
17  Done
18          STR     R1, Shifted             ; otherwise done - store result
19          STR     R3, Normal
20          SWI     &11
21
22          AREA    Data1, DATA
```

```
23
24  Table
25  *       DCD     &30001000               ; table of values to be tested
26  *       DCD     &00000001
27  *       DCD     &00000000
28          DCD     &C1234567
29
30          AREA    Result, DATA
31
32  Number  DCD     Table
33  Shifted DCB     0                       ; storage for shift
34          ALIGN
35  Normal  DCD     0                       ; storage for result
36
37          END
```

Algorithm 9.5b

```
Value ← M(Number)                    R0
Count ← 0                            R1
While Value[31] = 0
   Value ← Value ≪ 1
   Count ← Count + 1
End While
M(Shift) ← Count
M(Normal) ← Value
```

Program 9.5b: `normal2.s` — *Normalise a binary number*

File *loops/normal2.s* does not exist!

9.2 Problems

9.2.1 Checksum of data

Calculate the checksum of a series of 8-bit numbers. The length of the series is defined by the variable `LENGTH`. The label `START` indicates the start of the table. Store the checksum in the variable `CHECKSUM`. The checksum is formed by adding all the numbers in the list, ignoring the carry over (or overflow).

Note: Checksums are often used to ensure that data has been correctly read. A checksum calculated when reading the data is compared to a checksum that is stored with the data. If the two checksums do not agree, the system will usually indicate an error, or automatically read the data again.

Sample Problem:

Input:	`LENGTH`	00000003	(*Number of items*)
	`START`	28	(*Start of data table*)
		55	
		26	
Output:	`CHECKSUM`	$28 + 55 + 26$	(*Data Checksum*)
		= 00101000 (28)	
		+ 01010101 (55)	
		= 01111101 (7D)	
		+ 00100110 (26)	
		= 10100011 (A3)	

9.2.2 Number of Zero, Positive, and Negative numbers

Determine the number of zero, positive (most significant bit zero, but entire number not zero), and negative (most significant bit set) elements in a series of signed 32-bit numbers. The length of the series is defined by the variable `LENGTH` and the starting series of numbers start with the `START` label. Place the number of negative elements in variable `NUMNEG`, the number of zero elements in variable `NUMZERO` and the number of positive elements in variable `NUMPOS`.

Sample Problem:

Input:	`LENGTH`	6	(*Number of items*)
	`START`	76028326	(*Start of data table — Positive*)
		8D489867	(*Negative*)
		21202549	(*Positive*)
		00000000	(*Zero*)
		E605546C	(*Negative*)
		00000004	(*Positive*)
Output:	`NUMNEG`	2	(*2 negative numbers: 8D489867 and E605546C*)
	`NUMZERO`	1	(*1 zero value*)
	`NUMPOS`	3	(*3 positive numbers: 76028326, 21202549 and 00000004*)

9.2.3 Find Minimum

Find the smallest element in a series of unsigned bytes. The length of the series is defined by the variable `LENGTH` with the series starting at the `START` label. Store the minimum byte value in the `NUMMIN` variable.

Sample Problem:

Input:	`LENGTH`	5	(*Number of items*)
	`START`	65	(*Start of data table*)
		79	
		15	
		E3	
		72	
Output:	`NUMMIN`	15	(*Smallest of the five*)

9.2.4 Count 1 Bits

Determine the number of bits which are set in the 32-bit variable `NUM`, storing the result in the `NUMBITS` variable.

Sample Problem:

Input:	`NUM`	2866B794 = 0011 1000 0110 0110 1011 0111 1001 0100
Output:	`NUMBITS`	0F = 15

9.2.5 Find element with most 1 bits

Determine which element in a series of 32-bit numbers has the largest number of bits set. The length of the series is defined by the `LENGTH` variable and the series starts with the `START` label. Store the value with the most bits set in the `NUM` variable.

Sample Problem:

Input:	`LENGTH`	5	(*Number of items*)
	`START`	205A15E3	(0010 0000 0101 1010 0001 0101 1101 0011 — 13)
		256C8700	(0010 0101 0110 1100 1000 0111 0000 0000 — 11)
		295468F2	(0010 1001 0101 0100 0110 1000 1111 0010 — 14)
		29856779	(0010 1001 1000 0101 0110 0111 0111 1001 — 16)
		9147592A	(1001 0001 0100 0111 0101 1001 0010 1010 — 14)
Output:	`NUM`	29856779	(*Number with most 1-bits*)

10 Strings

Microprocessors often handle data which represents printed characters rather than numeric quantities. Not only do keyboards, printers, communications devices, displays, and computer terminals expect or provide character-coded data, but many instruments, test systems, and controllers also require data in this form. ASCII (American Standard Code for Information Interchange) is the most commonly used code, but others exist.

We use the standard seven-bit ASCII character codes, as shown in Table 10.1; the character code occupies the low-order seven bits of the byte, and the most significant bit of the byte holds a 0 or a parity bit.

10.1 Handling data in ASCII

Here are some principles to remember in handling ASCII-coded data:

- The codes for the numbers and letters form ordered sequences. Since the ASCII codes for the characters "0" through "9" are 30_{16} through 39_{16} you can convert a decimal digit to the equivalent ASCII characters (and ASCII to decimal) by simple adding the ASCII offset: 30_{16} = ASCII "0". Since the codes for letters (41_{16} through $5A_{16}$ and 61_{16} through $7A_{16}$) are in order, you can alphabetises strings by sorting them according to their numerical values.
- The computer does not distinguish between printing and non-printing characters. Only the I/0 devices make that distinction.
- An ASCII I/0 device handles data only in ASCII. For example, if you want an ASCII printer to print the digit "7", you must send it 37_{16} as the data; 07_{16} will ring the bell. Similarly, if a user presses the "9" key on an ASCII keyboard, the input data will be 39_{16}; 09_{16} is the tab key.
- Many ASCII devices do not use the entire character set. For example, devices may ignore many control characters and may not print lower-case letters.
- Despite the definition of the control characters many devices interpret them differently. For example they typically uses control characters in a special way to provide features such as cursor control on a display, and to allow software control of characteristics such as rate of data transmission, print width, and line length.
- Some widely used ASCII control characters are:

$0A_{16}$	LF	line feed
$0D_{16}$	CR	carriage return
08_{16}	BS	backspace
$7F_{16}$	DEL	rub out or delete character

- Each ASCII character occupies eight bits. This allows a large character set but is wasteful when only a few characters are actually being used. If, for example, the data consists entirely of decimal numbers, the ASCII format (allowing one digit per byte) requires twice as much

LSB	MSB 0	1	2	3	4	5	6	7
0	NUL	DLE	SP	0	@	P	‘	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	’	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	DEL

Control Characters			
NUL	Null	DLE	Data link escape
SOH	Start of heading	DC1	Device control 1
STX	Start of text	DC2	Device control 2
ETX	End of text	DC3	Device control 3
EOT	End of tx	DC4	Device control 4
ENQ	Enquiry	NAK	Negative ack
ACK	Acknowledge	SYN	Synchronous idle
BEL	Bell, or alarm	ETB	End of tx block
BS	Backspace	CAN	Cancel
HT	Horizontal tab	EM	End of medium
LF	Line feed	SUB	Substitute
VT	Vertical tab	ESC	Escape
FF	Form feed	FS	File separator
CR	Carriage return	GS	Group separator
SO	Shift out	RS	Record separator
SI	Shift in	US	Unit separator
SP	Space	DEL	Delete

Table 10.1: Hexadecimal ASCII Character Codes

storage, communications capacity, and processing time as the BCD format (allowing two digits per byte).

The assembler includes a feature to make character-coded data easy to handle, single quotation marks around a character indicate the character's ASCII value. For example,

```
MOV    R3, #'A'
```

is the same as

```
MOV    R3, #0x41
```

The first form is preferable for several reasons. It increases the readability of the instruction, it also avoids errors that may result from looking up a value in a table. The program does not depend on ASCII as the character set, since the assembler handles the conversion using whatever code has been designed for.

10.2 A string of characters

Individual characters on there own are not really all that helpful. As humans we need a string of characters in order to form meaningful text. In assembly programming it is normal to have to process one character at a time. However, the assembler does at least allow us to store a string of byes (characters) in a friendly manner with the DCB directive. For example, line 26 of program 10.1a is:

```
DCB    "Hello, World", CR
```

which will produce the following binary data:

Binary:	48	65	6C	6C	6F	2C	20	57	6F	72	6C	64	0D
Text:	H	e	l	l	o	,	SP	W	o	r	l	d	CR

Use table 10.1 to check that this is correct. In order to make the program just that little bit more readable, line 5 defines the label CR to have the value for a Carriage Return ($0D_{16}$).

There are three main methods for handling strings: Fixed Length, Terminated, and Counted. It is normal for a high level language to support just one method. C/C++ and Java all support the use of Zero-Terminated strings, while Pascal and Ada use counted strings. Although it is possible

to provide your own support for the alternative string type it is seldom done. A good programmer will use a mix of methods depending of the nature of the strings concerned.

10.2.1 Fixed Length Strings

A fixed length string is where the string is of a predefined and fixed size. For example, in a system where it is known that all strings are going to be ten characters in length, we can simply reserve 10 bytes for the string.

This has an immediate advantages in that the management of the strings is simple when compared to the alternative methods. For example we only need one label for an array of strings, and we can calculate the starting position of the n^{th} string by a simple multiplication.

This advantage is however also a major disadvantage. For example a persons name can be anything from two characters to any number of characters. Although it would be possible to reserve sufficient space for the longest of names this amount of memory would be required for all names, including the two letter ones. This is a significant waist of memory.

It would be possible to reserve just ten characters for each name. When a two letter name appears it would have to be padded out with spaces in order to make the name ten characters in length. When a name longer than ten characters appears it would have to be truncated down to just ten characters thus chopping off part of the name. This requires extra processing and is not entirely friendly to users who happen to have a long name.

When there is little memory and all the strings are known in advance it may be a good idea to use fixed length strings. For example, command driven systems tend to use a fixed length strings for the list of commands.

10.2.2 Terminated Strings

A terminated string is one that can be of any length and uses a special character to mark the end of the string, this character is known at the *sentinel*. For example program 10.1a uses the carriage return as it's sentinel.

Over the years several different sentinels have been used, these include \$ (26_{16}), EOT (End of Text – 04_{16}), CR (Carriage Return – $0D_{16}$), LF (Line Feed – $0A_{16}$) and NUL (No character – 00_{16}). Today the most commonly used sentinel is the NUL character, primarily because it is used by C/C++. The NUL character also has a good feeling about it, as it is represented by the value 0, has no other meaning and it is easier to detected than any other character. This is frequently referred to as a Null- or Zero-Terminated string or simply as an ASCIIZ string.

The terminated string has the advantage that it can be of any length. Processing the string is fairly simply, you enter into a loop processing each character at a time until you reach the sentinel. The disadvantage is that the sentinel character can not appear in the string. This is another reason why the NUL character is such a good choice for the sentinel.

10.2.3 Counted Strings

A counted string is one in which the first one or two byte holds the length of the string in characters. Thus a counted string can be of any number of characters up to the largest unsigned number that can be stored in the first byte/word.

A counted string may appear rather clumsy at first. Having the length of the string as a binary value has a distinct advantage over the terminated string. It allow the use of the counting instruc-

tions that have been included in many instruction sets. This means we can ignore the testing for a sentinel character and simply decrement our counter, this is a far faster method of working.

To scan through an array of strings we simply point to the first string, and add the length count to our pointer to obtain the start of the next string. For a terminated string we would have to scan for the sentinel for each string.

There are two disadvantages with the counted string. The string does have a maximum length, 255 characters or 64K depending on the size of the count value (8- or 16-bit). Although it is normally felt that 64K should be sufficient for most strings. The second disadvantage is their perceived complexity. Many people feel that the complexity of the counted string outweighs the speed advantage.

10.3 International Characters

As computing expands outside of the English speaking world we have to provide support for languages other than standard American. Many European languages use letters that are not available in standard ASCII, for example: œ, Œ, ø, Ø, æ, Æ, ł, Ł, ß, ¡, and ¿. This is particularly important when dealing with names: Ångstrøm, Karlstraße or Łukasiewicz.

The ASCII character set is not even capable of handling English correctly. When we borrow a word from another language we also use it's *diacritic marks* (or *accents*). For example I would rather see pâté on a menu rather than pate. ASCII does not provide support for such accents.

To overcome this limitation the international community has produced a new character encoding, known as *Unicode*. In Unicode the character code is two bytes long, the first byte indicates which *character set* the character comes from, while the second byte indicates the character position within the character set. The traditional ASCII character set is incorporated into Unicode as character set zero. In the revised `C` standard a new data type of `wchar` was defined to cater for this new "wide character".

While Unicode is sufficient to represent the characters from most modern languages, it is not sufficient to represent all the written languages of the world, ancient and modern. Hence an extended version, known as Unicode-32 is being developed where the character set is a 23-bit value (three bytes). Unicode is a subset of Unicode-32, while ASCII is a subset of Unicode.

Although we do not consider Unicode you should be aware of the problem of international character sets and the solution Unicode provides.

10.4 Program Examples

10.4.1 Length of a String of Characters

Program 10.1a: `strlencr.s` — *Find the length of a Carage Return terminated string*

```
1   ; Find the length of a CR terminated string
2
3           TTL     Ch6Ex1 - strlencr
4
5   CR      EQU     0x0D
6
7           AREA    Program, CODE, READONLY
8           ENTRY
9
10  Main
11          LDR     R0, =Data1              ; Load the address of the lookup table
```

```
12          EOR     R1, R1, R1              ; Clear R1 to store count
13  Loop
14          LDRB    R2, [R0], #1            ; Load the first byte into R2
15          CMP     R2, #CR                 ; Is it the terminator ?
16          BEQ     Done                    ; Yes => Stop loop
17          ADD     R1, R1, #1              ; No  => Increment count
18          BAL     Loop                    ; Read next char
19
20  Done
21          STR     R1, CharCount           ; Store result
22          SWI     &11
23
24          AREA    Data1, DATA
25
26  Table
27          DCB     "Hello, World", CR
28          ALIGN
29
30          AREA    Result, DATA
31  CharCount
32          DCB     0                       ; Storage for count
33
34          END
```

Program 10.1b: `strlen.s` — *Find the length of a null terminated string*

```
1   ; Find the length of a null terminated string
2
3           TTL     Ch6Ex1 - strlen
4           AREA    Program, CODE, READONLY
5           ENTRY
6
7   Main
8           LDR     R0, =Data1              ; Load the address of the lookup table
9           MOV     R1, #-1                 ; Start count at -1
10  Loop
11          ADD     R1, R1, #1              ; Increment count
12          LDRB    R2, [R0], #1            ; Load the first byte into R2
13          CMP     R2, #0                  ; Is it the terminator ?
14          BNE     Loop                    ; No => Next char
15
16          STR     R1, CharCount           ; Store result
17          SWI     &11
18
19          AREA    Data1, DATA
20
21  Table
22          DCB     "Hello, World", 0
23          ALIGN
24
25          AREA    Result, DATA
26  CharCount
27          DCB     0                       ; Storage for count
28
29          END
```

10.4.2 Find First Non-Blank Character

Program 10.2: `skipblanks.s` — *Find first non-blank*

```
1   *       find the length of a string
2
3           TTL     Ch6Ex3
4
```

```
5   Blank   EQU     " "
6           AREA    Program, CODE, READONLY
7           ENTRY
8
9   Main
10          ADR     R0, Data1               ;load the address of the lookup table
11          MOV     R1, #Blank              ;store the blank char in R1
12  Loop
13          LDRB    R2, [R0], #1            ;load the first byte into R2
14          CMP     R2, R1                  ;is it a blank
15          BEQ     Loop                    ;if so loop
16
17          SUB     R0, R0, #1              ;otherwise done - adjust pointer
18          STR     R0, Pointer             ;and store it
19          SWI     &11
20
21          AREA    Data1, DATA
22
23  Table
24          DCB     "      7   "
25          ALIGN
26
27          AREA    Result, DATA
28  Pointer DCD     0                       ;storage for count
29          ALIGN
30
31          END
```

10.4.3 Replace Leading Zeros with Blanks

Program 10.3: `padzeros.s` — *Supress leading zeros in a string*

```
1   *       supress leading zeros in a string
2
3           TTL     Ch6Ex4
4
5   Blank   EQU     ' '
6   Zero    EQU     '0'
7           AREA    Program, CODE, READONLY
8           ENTRY
9
10  Main
11          LDR     R0, =Data1              ;load the address of the lookup table
12          MOV     R1, #Zero               ;store the zero char in R1
13          MOV     R3, #Blank              ;and the blank char in R3
14  Loop
15          LDRB    R2, [R0], #1            ;load the first byte into R2
16          CMP     R2, R1                  ;is it a zero
17          BNE     Done                    ;if not, done
18
19          SUB     R0, R0, #1              ;otherwise adjust the pointer
20          STRB    R3, [R0]                ;and store it blank char there
21          ADD     R0, R0, #1              ;otherwise adjust the pointer
22          BAL     Loop                    ;and loop
23
24  Done
25          SWI     &11                     ;all done
26
27          AREA    Data1, DATA
28
29  Table
30          DCB     "000007000"
31          ALIGN
32
```

```
33          AREA    Result, DATA
34  Pointer DCD     0                       ;storage for count
35          ALIGN
36
37          END
```

10.4.4 Add Even Parity to ASCII Chatacters

Program 10.4: `setparity.s` — *Set the parity bit on a series of characters store the amended string in Result*

```
1   ; Set the parity bit on a series of characters store the amended string in Result
2
3           TTL     Ch6Ex5
4
5           AREA    Program, CODE, READONLY
6           ENTRY
7
8   Main
9           LDR     R0, =Data1              ;load the address of the lookup table
10          LDR     R5, =Pointer
11          LDRB    R1, [R0], #1            ;store the string length in R1
12          CMP     R1, #0
13          BEQ     Done                    ;nothing to do if zero length
14  MainLoop
15          LDRB    R2, [R0], #1            ;load the first byte into R2
16          MOV     R6, R2                  ;keep a copy of the original char
17          MOV     R2, R2, LSL #24         ;shift so that we are dealing with msb
18          MOV     R3, #0                  ;zero the bit counter
19          MOV     R4, #7                  ;init the shift counter
20
21  ParLoop
22          MOVS    R2, R2, LSL #1          ;left shift
23          BPL     DontAdd                 ;if msb is not a one bit, branch
24          ADD     R3, R3, #1              ;otherwise add to bit count
25  DontAdd
26          SUBS    R4, R4, #1              ;update shift count
27          BNE     ParLoop                 ;loop if still bits to check
28          TST     R3, #1                  ;is the parity even
29          BEQ     Even                    ;if so branch
30          ORR     R6, R6, #0x80           ;otherwise set the parity bit
31          STRB    R6, [R5], #1            ;and store the amended char
32          BAL     Check
33  Even    STRB    R6, [R5], #1            ;store the unamended char if even pty
34  Check   SUBS    R1, R1, #1              ;decrement the character count
35          BNE     MainLoop
36
37  Done    SWI     &11
38
39          AREA    Data1, DATA
40
41  Table   DCB     6                       ;data table starts with byte length of string
42          DCB     0x31                    ;the string
43          DCB     0x32
44          DCB     0x33
45          DCB     0x34
46          DCB     0x35
47          DCB     0x36
48
49          AREA    Result, DATA
50          ALIGN
51  Pointer DCD     0                       ;storage for parity characters
52
53          END
```

10.4.5 Pattern Match

Program 10.5a: `cstrcmp.s` — *Compare two counted strings for equality*

```
1   *       compare two counted strings for equality
2
3           TTL     Ch6Ex6
4
5           AREA    Program, CODE, READONLY
6           ENTRY
7
8   Main
9           LDR     R0, =Data1              ;load the address of the lookup table
10          LDR     R1, =Data2
11          LDR     R2, Match               ;assume strings not equal - set to -1
12          LDR     R3, [R0], #4            ;store the first string length in R3
13          LDR     R4, [R1], #4            ;store the second string length in R4
14          CMP     R3, R4
15          BNE     Done                    ;if they are different lengths,
16                                          ;they can't be equal
17          CMP     R3, #0                  ;test for zero length if both are
18          BEQ     Same                    ;zero length, nothing else to do
19
20  *       if we got this far, we now need to check the string char by char
21  Loop
22          LDRB    R5, [R0], #1            ;character of first string
23          LDRB    R6, [R1], #1            ;character of second string
24          CMP     R5, R6                  ;are they the same
25          BNE     Done                    ;if not the strings are different
26          SUBS    R3, R3, #1              ;use the string length as a counter
27          BEQ     Same                    ;if we got to the end of the count
28                                          ;the strings are the same
29          B       Loop                    ;not done, loop
30
31  Same    MOV     R2, #0                  ;clear the -1 from match (0 = match)
32  Done    STR     R2, Match               ;store the result
33          SWI     &11
34
35          AREA    Data1, DATA
36  Table1  DCD     3                       ;data table starts with byte length of string
37          DCB     "CAT"                   ;the string
38
39          AREA    Data2, DATA
40  Table2  DCD     3                       ;data table starts with byte length of string
41          DCB     "CAT"                   ;the string
42
43          AREA    Result, DATA
44          ALIGN
45  Match   DCD     &FFFF                   ;storage for parity characters
46
47          END
```

Program 10.5b: `strcmp.s` — *Compare null terminated strings for equality assume that we have no knowledge of the data structure so we must assess the individual strings*

```
1   ; Compare two null terminated strings for equality
2
3           TTL     Ch6Ex7
4
5           AREA    Program, CODE, READONLY
6           ENTRY
7
```

```
 8  Main
 9          LDR     R0, =Data1              ;load the address of the lookup table
10          LDR     R1, =Data2
11          LDR     R2, Match               ;assume strings not equal, set to -1
12          MOV     R3, #0                  ;init register
13          MOV     R4, #0
14  Count1
15          LDRB    R5, [R0], #1            ;load the first byte into R5
16          CMP     R5, #0                  ;is it the terminator
17          BEQ     Count2                  ;if not, Loop
18          ADD     R3, R3, #1              ;increment count
19          BAL     Count1
20  Count2
21          LDRB    R5, [R1], #1            ;load the first byte into R5
22          CMP     R5, #0                  ;is it the terminator
23          BEQ     Next                    ;if not, Loop
24          ADD     R4, R4, #1              ;increment count
25          BAL     Count2
26
27  Next    CMP     R3, R4
28          BNE     Done                    ;if they are different lengths,
29                                          ;they can't be equal
30          CMP     R3, #0                  ;test for zero length if both are
31          BEQ     Same                    ;zero length, nothing else to do
32          LDR     R0, =Data1              ;need to reset the lookup table
33          LDR     R1, =Data2
34
35  *       if we got this far, we now need to check the string char by char
36  Loop
37          LDRB    R5, [R0], #1            ;character of first string
38          LDRB    R6, [R1], #1            ;character of second string
39          CMP     R5, R6                  ;are they the same
40          BNE     Done                    ;if not the strings are different
41          SUBS    R3, R3, #1              ;use the string length as a counter
42          BEQ     Same                    ;if we got to the end of the count
43                                          ;the strings are the same
44          BAL     Loop                    ;not done, loop
45
46  Same
47          MOV     R2, #0                  ;clear the -1 from match (0 = match)
48  Done
49          STR     R2, Match               ;store the result
50          SWI     &11
51
52          AREA    Data1, DATA
53  Table1  DCB     "Hello, World", 0       ;the string
54          ALIGN
55
56          AREA    Data2, DATA
57  Table2  DCB     "Hello, worl", 0        ;the string
58
59          AREA    Result, DATA
60          ALIGN
61  Match   DCD     &FFFF                   ;flag for match
62
63          END
```

10.5 Problems

10.5.1 Length of a Teletypewriter Message

Determine the length of an ASCII message. All characters are 7-bit ASCII with MSB = 0. The string of characters in which the message is embedded has a starting address which is contained in the `START` variable. The message itself starts with an ASCII *STX* (Start of Text) character (02_{16}) and ends with *ETX* (End of Text) character (03_{16}). Save the length of the message, the number of characters between the *STX* and the *ETX* markers (but not including the markers) in the `LENGTH` variable.

Sample Problem:

Input:	`START`	`String`	(*Location of string*)
	String	02	(STX — *Start Text*)
		47	("`G`")
		4F	("`O`")
		03	(ETX — *End Text*)
Output:	`LENGTH`	02	("GO")

10.5.2 Find Last Non-Blank Character

Search a string of ASCII characters for the last non-blank character. Starting address of the string is contained in the `START` variable and the string ends with a carriage return character ($0D_{16}$). Place the address of the last non-blank character in the `POINTER` variable.

Sample Problems:

		Test A	Test B
Input:	`START`	`String`	`String`
	String	37 ("`7`")	41 ("`A`")
		0D (CR)	20 (Space)
			48 ("`H`")
			41 ("`A`")
			54 ("`T`")
			20 (Space)
			20 (Space)
			0D (CR)
Output:	`POINTER`	*First Char*	*Fourth Char*

10.5.3 Truncate Decimal String to Integer Form

Edit a string of ASCII decimal characters by replacing all digits to the right of the decimal point with ASCII blanks (20_{16}). The starting address of the string is contained in the `START` variable and the string is assumed to consist entirely of ASCII-coded decimal digits and a possible decimal point ($2E_{16}$). The length of the string is stored in the `LENGTH` variable. If no decimal point appears in the string, assume that the decimal point is at the far right.

Sample Problems:

		Test A	Test B
Input:	`START`	`String`	`String`
	`LENGTH`	4	3
	String	37 (“7”)	36 (“6”)
		2E (“.”)	37 (“7”)
		38 (“8”)	31 (“1”)
		31 (“1”)	
Output:		37 (“7”)	36 (“6”)
		2E (“.”)	37 (“7”)
		20 (Space)	31 (“1”)
		20 (Space)	

Note that in the second case (Test B) the output is unchaged, as the number is assumed to be “671.”.

10.5.4 Check Even Parity and ASCII Characters

Cheek for even parity in a string of ASCII characters. A string's starting address is contained in the `START` variable. The first word of the string is its length which is followed by the string itself. If the parity of all the characters in the string are correct, clear the `PARITY` variable; otherwise place all ones ($FFFFFFFF_{16}$) into the variable.

Sample Problems:

		Test A	Test B
Input:	`START`	`String`	`String`
	String	3	03
		B1 (1011 0001)	B1 (1011 0001)
		B2 (1011 0010)	B6 (1011 0110)
		33 (0011 0011)	33 (0011 0011)
Output:	`PARITY`	00000000 (True)	FFFFFFFF (False)

10.5.5 String Comparison

Compare two strings of ASCII characters to see which is larger (that is, which follows the other in alphabetical ordering). Both strings have the same length as defined by the `LENGTH` variable. The strings' starting addresses are defined by the `START1` and `START2` variables. If the string defined by `START1` is greater than or equal to the other string, clear the `GREATER` variable; otherwise set the variable to all ones ($FFFFFFFF_{16}$).

Sample Problems:

		Test A	Test B	Test C
Input:	`LENGTH`	3	3	3
	`START1`	`String1`	`String1`	`String1`
	`START2`	`String2`	`String2`	`String2`
	String1	43 (“C”)	43 (“C”)	43 (“C”)
		41 (“A”)	41 (“A”)	41 (“A”)
		54 (“T”)	54 (“T”)	54 (“T”)
	String2	42 (“B”)	52 (“C”)	52 (“C”)
		41 (“A”)	41 (“A”)	55 (“U”)
		54 (“T”)	54 (“T”)	54 (“T”)
Output:	`GREATER`	00000000 (CAT > BAT)	00000000 (CAT = CAT)	FFFFFFFF (CAT < CUT)

11 Code Conversion

Code conversion is a continual problem in microcomputer applications. Peripherals provide data in ASCII, BCD or various special codes. The microcomputer must convert the data into some standard form for processing. Output devices may require data in ASCII, BCD, seven-segment or other codes. Therefore, the microcomputer must convert the results to the proper form after it completes the processing.

There are several ways to approach code conversion:

1. Some conversions can easily be handled by algorithms involving arithmetic or logical functions. The program may, however, have to handle special cases separately.
2. More complex conversions can be handled with lookup tables. The lookup table method requires little programming and is easy to apply. However the table may occupy a large amount of memory if the range of input values is large.
3. Hardware is readily available for some conversion tasks. Typical examples are decoders for BCD to seven-segment conversion and Universal Asynchronous Receiver/Transmitters (UARTs) for conversion between parallel and serial formats.

In most applications, the program should do as much as possible of the code conversion work. Most code conversions are easy to program and require little execution time.

11.1 Program Examples

11.1.1 Hexadecimal to ASCII

Program 11.1a: `nibtohex.s` — *Convert a single hex digit to its ASCII equivalent*

```
1   *       convert a single hex digit to its ASCII equivalent
2
3           TTL     Ch7Ex1
4
5           AREA    Program, CODE, READONLY
6           ENTRY
7
8   Main
9           LDR     R0, Digit               ;load the digit
10          LDR     R1, =Result             ;load the address for the result
11          CMP     R0, #0xA                ;is the number < 10 decimal
12          BLT     Add_0                   ;then branch
13
14          ADD     R0, R0, #"A"-"0"-0xA    ;add offset for 'A' to 'F'
15  Add_0
16          ADD     R0, R0, #"0"            ;convert to ASCII
17          STR     R0, [R1]                ;store the result
18          SWI     &11
19
20          AREA    Data1, DATA
```

```
21  Digit
22          DCD     &0C                     ;the hex digit
23
24          AREA    Data2, DATA
25  Result  DCD     0                       ;storage for result
26
27          END
```

Program 11.1b: `wordtohex.s` — *Convert a 32 bit hexadecimal number to an ASCII string and output to the terminal*

```
1   *       now something a little more adventurous - convert a 32 bit
2   *       hexadecimal number to an ASCII string and output to the terminal
3
4           TTL     Ch7Ex2
5
6           AREA    Program, CODE, READONLY
7           ENTRY
8   Mask    EQU     0x0000000F
9
10  start
11          LDR     R1, Digit               ;load the digit
12          MOV     R4, #8                  ;init counter
13          MOV     R5, #28                 ;control right shift
14  MainLoop
15          MOV     R3, R1                  ;copy original word
16          MOV     R3, R3, LSR R5          ;right shift the correct number of bits
17          SUB     R5, R5, #4              ;reduce the bit shift
18          AND     R3, R3, #Mask           ;mask out all but the ls nibble
19          CMP     R3, #0xA                ;is the number < 10 decimal
20          BLT     Add_0                   ;then branch
21
22          ADD     R3, R3, #"A"-"0"-0xA    ;add offset for 'A' to 'F'
23
24  Add_0   ADD     R3, R3, #"0"            ;convert to ASCII
25          MOV     R0, R3                  ;prepare to output
26          SWI     &0                      ;output to console
27          SUBS    R4, R4, #1              ;decrement counter
28          BNE     MainLoop
29
30          MOV     R0, #&0D                ;add a CR character
31          SWI     &0                      ;output it
32          SWI     &11                     ;all done
33
34          AREA    Data1, DATA
35  Digit   DCD     &DEADBEEF               ;the hex word
36
37          END
```

11.1.2 Decimal to Seven-Segment

Program 11.2: `nibtoseg.s` — *Convert a decimal number to seven segment binary*

```
1   *       convert a decimal number to seven segment binary
2
3           TTL     Ch7Ex3
4
5           AREA    Program, CODE, READONLY
6           ENTRY
7
8   Main
9           LDR     R0, =Data1              ;load the start address of the table
10          EOR     R1, R1, R1              ;clear register for the code
11          LDRB    R2, Digit               ;get the digit to encode
```

```
12          CMP     R2, #9                  ;is it a valid digit?
13          BHI     Done                    ;clear the result
14
15          ADD     R0, R0, R2              ;advance the pointer
16          LDRB    R1, [R0]                ;and get the next byte
17  Done
18          STR     R1, Result              ;store the result
19          SWI     &11                     ;all done
20
21          AREA    Data1, DATA
22  Table   DCB     &3F                     ;the binary conversions table
23          DCB     &06
24          DCB     &5B
25          DCB     &4F
26          DCB     &66
27          DCB     &6D
28          DCB     &7D
29          DCB     &07
30          DCB     &7F
31          DCB     &6F
32          ALIGN
33
34          AREA    Data2, DATA
35  Digit   DCB     &05                     ;the number to convert
36          ALIGN
37
38          AREA    Data3, DATA
39  Result  DCD     0                       ;storage for result
40
41          END
```

11.1.3 ASCII to Decimal

Program 11.3: `dectonib.s` — *Convert an ASCII numeric character to decimal*

```
1   *       convert an ASCII numeric character to decimal
2
3           TTL     Ch7Ex4
4
5           AREA    Program, CODE, READONLY
6           ENTRY
7
8   Main
9           MOV     R1, #-1                 ;set -1 as error flag
10          LDRB    R0, Char                ;get the character
11          SUBS    R0, R0, #"0"            ;convert and check if character is < 0
12          BCC     Done                    ;if so do nothing
13          CMP     R0, #9                  ;check if character is > 9
14          BHI     Done                    ;if so do nothing
15          MOV     R1, R0                  ;otherwise....
16  Done
17          STR     R1, Result              ;.....store the decimal no
18          SWI     &11                     ;all done
19
20          AREA    Data1, DATA
21  Char    DCB     &37                     ;ASCII representation of 7
22          ALIGN
23
24          AREA    Data2, DATA
25  Result  DCD     0                       ;storage for result
26
27          END
```

11.1.4 Binary-Coded Decimal to Binary

Program 11.4a: `ubcdtohalf.s` — *Convert an unpacked BCD number to binary*

```
1   *       convert an unpacked BCD number to binary
2
3           TTL     Ch7Ex5
4
5           AREA    Program, CODE, READONLY
6           ENTRY
7
8   Main
9           LDR     R0, =BCDNum             ;load address of BCD number
10          MOV     R5, #4                  ;init counter
11          MOV     R1, #0                  ;clear result register
12          MOV     R2, #0                  ;and final register
13
14  Loop
15          ADD     R1, R1, R1              ;multiply by 2
16          MOV     R3, R1
17          MOV     R3, R3, LSL #2          ;mult by 8 (2 x 4)
18          ADD     R1, R1, R3              ;= mult by 10
19
20          LDRB    R4, [R0], #1            ;load digit and incr address
21          ADD     R1, R1, R4              ;add the next digit
22          SUBS    R5, R5, #1              ;decr counter
23          BNE     Loop                    ;if counter != 0, loop
24
25          STR     R1, Result              ;store the result
26          SWI     &11                     ;all done
27
28          AREA    Data1, DATA
29  BCDNum  DCB     &02,&09,&07,&01         ;an unpacked BCD number
30          ALIGN
31
32          AREA    Data2, DATA
33  Result  DCD     0                       ;storage for result
34
35          END
```

Program 11.4b: `ubcdtohalf2.s` — *Convert an unpacked BCD number to binary using MUL*

```
1   *       convert an unpacked BCD number to binary using MUL
2
3           TTL     Ch7Ex6
4
5           AREA    Program, CODE, READONLY
6           ENTRY
7
8   Main
9           LDR     R0, =BCDNum             ;load address of BCD number
10          MOV     R5, #4                  ;init counter
11          MOV     R1, #0                  ;clear result register
12          MOV     R2, #0                  ;and final register
13          MOV     R7, #10                 ;multiplication constant
14
15  Loop
16          MOV     R6, R1
17          MUL     R1, R6, R7              ;mult by 10
18          LDRB    R4, [R0], #1            ;load digit and incr address
19          ADD     R1, R1, R4              ;add the next digit
20          SUBS    R5, R5, #1              ;decr counter
21          BNE     Loop                    ;if count != 0, loop
22
23          STR     R1, Result              ;store the result
24          SWI     &11                     ;all done
```

```
25
26          AREA    Data1, DATA
27  BCDNum  DCB     &02,&09,&07,&01         ;an unpacked BCD number
28          ALIGN
29
30          AREA    Data2, DATA
31  Result  DCD     0                       ;storage for result
32
33          END
```

11.1.5 Binary Number to ASCII String

Program 11.5: `halftobin.s` — *Store a 16bit binary number as an ASCII string of '0's and '1's*

```
1   *       store a 16bit binary number as an ASCII string of '0's and '1's
2
3           TTL     Ch7Ex7
4           AREA    Program, CODE, READONLY
5           ENTRY
6
7   Main
8           LDR     R0, =String             ;load startr address of string
9           ADD     R0, R0, #16             ;adjust for length of string
10          LDRB    R6, String              ;init counter
11          MOV     R2, #"1"                ;load character '1' to register
12          MOV     R3, #"0"
13          LDR     R1, Number              ;load the number to process
14
15  Loop
16          MOVS    R1, R1, ROR #1          ;rotate right with carry
17          BCS     Loopend                 ;if carry set branch (LSB was a '1' bit)
18          STRB    R3, [R0], #-1           ;otherwise store "0"
19          BAL     Decr                    ;and branch to counter code
20  Loopend
21          STRB    R2, [R0], #-1           ;store a "1"
22  Decr
23          SUBS    R6, R6, #1              ;decrement counter
24          BNE     Loop                    ;loop while not 0
25
26          SWI     &11
27
28          AREA    Data1, DATA
29  Number  DCD     &31D2                   ;a 16 bit binary number number
30          ALIGN
31
32          AREA    Data2, DATA
33  String  DCB     16                      ;storage for result
34          ALIGN
35
36          END
```

11.2 Problems

11.2.1 ASCII to Hexadecimal

Convert the contents of the `A_DIGIT` variable from an ASCII character to a hexadecimal digit and store the result in the `H_DIGIT` variable. Assume that `A_DIGIT` contains the ASCII representation of a hexadecimal digit (7 bits with MSB=0).

		7	6	5	4	3	2	1	0
		0	g	f	e	d	c	b	a
0	3F	0	0	1	1	1	1	1	1
1	06	0	0	0	0	0	1	1	0
2	5B	0	1	0	1	1	0	1	1
3	4F	0	1	0	0	1	1	1	1
4	66	0	1	1	0	0	1	1	0
5	6D	0	1	1	0	1	1	0	1
6	7D	0	1	1	1	1	1	0	1
7	07	0	0	0	0	0	1	1	1
8	7F	0	1	1	1	1	1	1	1
9	6F	0	1	1	0	1	1	1	1
A	77	0	1	1	1	0	1	1	1
B	7C	0	1	1	1	1	1	0	0
C	3A	0	0	1	1	1	0	0	1
D	5E	0	1	0	1	1	1	1	0
E	7A	0	1	1	1	1	0	0	1
F	71	0	1	1	1	0	0	0	1

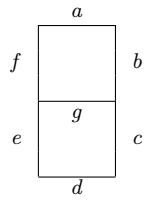


Figure 11.1: Seven-Segment Display

Sample Problems:

		Test A	Test B
Input:	`A_DIGIT`	43 (“C”)	36 (“6”)
Output:	`H_DIGIT`	0C	06

11.2.2 Seven-Segment to Decimal

Convert the contents of the `CODE` variable from a seven-segment code to a decimal number and store the result in the `NUMBER` variable. If `CODE` does not contain a valid seven-segment code, set `NUMBER` to FF_{16}. Use the seven-segment table given in Figure 11.1 and try to match codes.

Sample Problems:

		Test A	Test B
Input:	`CODE`	4F	28
Output:	`NUMBER`	03	FF

11.2.3 Decimal to ASCII

Convert the contents of the variable `DIGIT` from decimal digit to an ASCII character and store the result in the variable `CHAR`. If the number in `DIGIT` is not a decimal digit, set the contents of `CHAR` to an ASCII space (20_{16}).

Sample Problems:

		Test A	Test B
Input:	`DIGIT`	07	55
Output:	`CHAR`	37 (“7”)	20 (Space)

11.2.4 Binary to Binary-Coded-Decimal

Convert the contents of the variable `NUMBER` to four BCD digits in the `STRING` variable. The 32-bit number in `NUMBER` is unsigned and less than 10,000.

Sample Problem:

Input:	NUMBER	1C52	(7250_{10})
Output:	STRING	07	(“7”)
		02	(“2”)
		05	(“5”)
		00	(“0”)

11.2.5 Packed Binary-Coded-Decimal to Binary String

Convert the eight digit packed binary-coded-decimal number in the BCDNUM variable into a 32-bit number in a NUMBER variable.

Sample Problem:

Input:	BCDNUM	92529679
Output:	NUMBER	$0583E409_16$ (92529679_10)

11.2.6 ASCII string to Binary number

Convert the eight ASCII characters in the variable STRING to an 8-bit binary number in the variable NUMBER. Clear the byte variable ERROR if all the ASCII characters are either ASCII “1” or ASCII “0”; otherwise set ERROR to all ones (FF_{16}).

Sample Problems:

		Test A		Test B	
Input:	STRING	31	(“1”)	31	(“1”)
		31	(“1”)	31	(“1”)
		30	(“0”)	30	(“0”)
		31	(“1”)	31	(“1”)
		30	(“0”)	30	(“0”)
		30	(“0”)	37	(“7”)
		31	(“1”)	31	(“1”)
		30	(“0”)	30	(“0”)
Output:	NUMBER	D2	(1101 0010)	00	(*Valid*)
	ERROR	0	(*No Error*)	FF	(*Error*)

12 Arithmetic

Much of the arithmetic in some microprocessor applications consists of multiple-word binary or decimal manipulations. The processor provides for decimal addition and subtraction, but does not provide for decimal multiplication or division, you must implement these operations with sequences of instruction.

Most processors provide for both signed and unsigned binary arithmetic. Signed numbers are represented in two's complement form. This means that the operations of addition and subtraction are the same whether the numbers are signed or unsigned.

Multiple-precision binary arithmetic requires simple repetitions of the basic instructions. The Carry flag transfers information between words. It is set when an addition results in a carry or a subtraction results in a borrow. Add with Carry and Subtract with Carry use this information from the previous arithmetic operation.

Decimal arithmetic is a common enough task for microprocessors that most have special instructions for this purpose. These instructions may either perform decimal operations directly or correct the results of binary operations to the proper decimal form. Decimal arithmetic is essential in such applications as point-of-sale terminals, check processors, order entry systems, and banking terminals.

You can implement decimal multiplication and division as series of additions and subtractions, respectively. Extra storage must be reserved for results, since a multiplication produces a result twice as long as the operands. A division contracts the length of the result. Multiplications and divisions are time-consuming when done in software because of the repeated operations that are necessary.

12.1 Program Examples

12.1.2 64-Bit Addition

Program 12.2: `add64.s` — *64 Bit Addition*

```
1   *       64 bit addition
2
3           TTL     64 bit addition
4           AREA    Program, CODE, READONLY
5           ENTRY
6
7   Main
8           LDR     R0, =Value1             ; Pointer to first value
9           LDR     R1, [R0]                ; Load first part of value1
10          LDR     R2, [R0, #4]            ; Load lower part of value1
11          LDR     R0, =Value2             ; Pointer to second value
12          LDR     R3, [R0]                ; Load upper part of value2
13          LDR     R4, [R0, #4]            ; Load lower part of value2
14          ADDS    R6, R2, R4              ; Add lower 4 bytes and set carry flag
```

```
15          ADC     R5, R1, R3              ; Add upper 4 bytes including carry
16          LDR     R0, =Result             ; Pointer to Result
17          STR     R5, [R0]                ; Store upper part of result
18
19          STR     R6, [R0, #4]            ; Store lower part of result
20          SWI     &11
21
22  Value1  DCD     &12A2E640, &F2100123    ; Value to be added
23  Value2  DCD     &001019BF, &40023F51    ; Value to be added
24  Result  DCD     0                       ; Space to store result
25          END
```

12.1.3 Decimal Addition

Program 12.3: `addbcd.s` — *Add two packed BCD numbers to give a packed BCD result*

```
1   *       add two packed BCD numbers to give a packed BCD result
2
3           TTL     Ch8Ex3
4           AREA    Program, CODE, READONLY
5           ENTRY
6
7   Mask    EQU     0x0000000F
8
9   Main
10          LDR     R0, =Result             ;address for storage
11          LDR     R1, BCDNum1             ;load the first BCD number
12          LDR     R2, BCDNum2             ;and the second
13          LDRB    R8, Length              ;init counter
14          ADD     R0, R0, #3              ;adjust for offset
15          MOV     R5, #0                  ;carry
16
17  Loop
18          MOV     R3, R1                  ;copy what is left in the data register
19          MOV     R4, R2                  ;and the other number
20          AND     R3, R3, #Mask           ;mask out everything except low order nibble
21          AND     R4, R4, #Mask           ;mask out everything except low order nibble
22          MOV     R1, R1, LSR #4          ;shift the original number one nibble
23          MOV     R2, R2, LSR #4          ;shift the original number one nibble
24          ADD     R6, R3, R4              ;add the digits
25          ADD     R6, R6, R5              ;and the carry
26          CMP     R6, #0xA                ;is it over 10?
27          BLT     RCarry1                 ;if not, reset the carry to 0
28          MOV     R5, #1                  ;otherwise set the carry
29          SUB     R6, R6, #0xA            ;and subtract 10
30          B       Next
31  RCarry1
32          MOV     R5, #0                  ;carry reset to 0
33
34  Next
35          MOV     R3, R1                  ;copy what is left in the data register
36          MOV     R4, R2                  ;and the other number
37          AND     R3, R3, #Mask           ;mask out everything except low order nibble
38          AND     R4, R4, #Mask           ;mask out everything except low order nibble
39          MOV     R1, R1, LSR #4          ;shift the original number one nibble
40          MOV     R2, R2, LSR #4          ;shift the original number one nibble
41          ADD     R7, R3, R4              ;add the digits
42          ADD     R7, R7, R5              ;and the carry
43          CMP     R7, #0xA                ;is it over 10?
44          BLT     RCarry2                 ;if not, reset the carry to 0
45          MOV     R5, #1                  ;otherwise set the carry
46          SUB     R7, R7, #0xA            ;and subtract 10
47          B       Loopend
48
```

```
49  RCarry2
50          MOV     R5, #0                  ;carry reset to 0
51  Loopend
52          MOV     R7, R7, LSL #4          ;shift the second digit processed to the left
53          ORR     R6, R6, R7              ;and OR in the first digit to the ls nibble
54          STRB    R6, [R0], #-1           ;store the byte, and decrement address
55          SUBS    R8, R8, #1              ;decrement loop counter
56          BNE     Loop                    ;loop while > 0
57          SWI     &11
58
59          AREA    Data1, DATA
60  Length  DCB     &04
61          ALIGN
62  BCDNum1 DCB     &36, &70, &19, &85      ;an 8 digit packed BCD number
63
64          AREA    Data2, DATA
65  BCDNum2 DCB     &12, &66, &34, &59      ;another 8 digit packed BCD number
66
67          AREA    Data3, DATA
68  Result  DCD     0                       ;storage for result
69
70          END
```

12.1.4 Multiplication

16-Bit

Program 12.4a: `mul16.s` — *16 bit binary multiplication*

```
1   *       16 bit binary multiplication
2
3           TTL     Ch8Ex1
4           AREA    Program, CODE, READONLY
5           ENTRY
6
7   Main
8           LDR     R0, Number1             ;load first number
9           LDR     R1, Number2             ;and second
10          MUL     R0, R1, R0              ;x:= y * x
11  *       MUL     R0, R0, R1              ;won't work - not allowed
12          STR     R0, Result
13
14          SWI     &11                     ;all done
15
16          AREA    Data1, DATA
17  Number1 DCD     &706F                   ;a 16 bit binary number
18  Number2 DCD     &0161                   ;another
19          ALIGN
20
21          AREA    Data2, DATA
22  Result  DCD     0                       ;storage for result
23          ALIGN
24
25          END
```

32-Bit

Program 12.4b: `mul32.s` — *Multiply two 32 bit number to give a 64 bit result (corrupts R0 and R1)*

```
1   *       multiply two 32 bit number to give a 64 bit result
2   *       (corrupts R0 and R1)
3
4           TTL     Ch8Ex4
5           AREA    Program, CODE, READONLY
6           ENTRY
7
8   Main
9           LDR     R0, Number1             ;load first number
10          LDR     R1, Number2             ;and second
11          LDR     R6, =Result             ;load the address of result
12          MOV     R5, R0, LSR #16         ;top half of R0
13          MOV     R3, R1, LSR #16         ;top half of R1
14          BIC     R0, R0, R5, LSL #16     ;bottom half of R0
15          BIC     R1, R1, R3, LSL #16     ;bottom half of R1
16          MUL     R2, R0, R1              ;partial result
17          MUL     R0, R3, R0              ;partial result
18          MUL     R1, R5, R1              ;partial result
19          MUL     R3, R5, R3              ;partial result
20          ADDS    R0, R1, R0              ;add middle parts
21          ADDCS   R3, R3, #&10000         ;add in any carry from above
22          ADDS    R2, R2, R0, LSL #16     ;LSB 32 bits
23          ADC     R3, R3, R0, LSR #16     ;MSB 32 bits
24
25          STR     R2, [R6]                ;store LSB
26          ADD     R6, R6, #4              ;increment pointer
27          STR     R3, [R6]                ;store MSB
28          SWI     &11                     ;all done
29
30          AREA    Data1, DATA
31  Number1 DCD     &12345678               ;a 16 bit binary number
32  Number2 DCD     &ABCDEF01               ;another
33          ALIGN
34
35          AREA    Data2, DATA
36  Result  DCD     0                       ;storage for result
37          ALIGN
38
39          END
```

12.1.5 32-Bit Binary Divide

Program 12.5: `divide.s` — *Divide a 32 bit binary no by a 16 bit binary no store the quotient and remainder there is no 'DIV' instruction in ARM!*

```
1   *       divide a 32 bit binary no by a 16 bit binary no
2   *       store the quotient and remainder
3   *       there is no 'DIV' instruction in ARM!
4
5           TTL     Ch8Ex2
6           AREA    Program, CODE, READONLY
7           ENTRY
8
9   Main
10          LDR     R0, Number1             ;load first number
11          LDR     R1, Number2             ;and second
12          MOV     R3, #0                  ;clear register for quotient
13  Loop
14          CMP     R1, #0                  ;test for divide by 0
15          BEQ     Err
16          CMP     R0, R1                  ;is the divisor less than the dividend?
17          BLT     Done                    ;if so, finished
18          ADD     R3, R3, #1              ;add one to quotient
19          SUB     R0, R0, R1              ;take away the number you first thought of
20          B       Loop                    ;and loop
```

```
21  Err
22          MOV     R3, #0xFFFFFFFF         ;error flag (-1)
23  Done
24          STR     R0, Remain              ;store the remainder
25          STR     R3, Quotient            ;and the quotient
26          SWI     &11                     ;all done
27
28          AREA    Data1, DATA
29  Number1 DCD     &0075CBB1               ;a 16 bit binary number
30  Number2 DCD     &0141                   ;another
31          ALIGN
32
33          AREA    Data2, DATA
34  Quotient DCD    0                       ;storage for result
35  Remain  DCD     0                       ;storage for remainder
36          ALIGN
37
38          END
```

12.2 Problems

12.2.1 Multiple precision Binary subtraction

Subtract one multiple-word number from another. The length in words of both numbers is in the `LENGTH` variable. The numbers themselves are stored (most significant bits First) in the variables `NUM1` and `NUM2` respectively. Subtract the number in `NUM2` from the one in `NUM1`. Store the difference in `NUM1`.

Sample Problem:

Input:	`LENGTH`	3	(*Number of words in each number*)
	`NUM1`	2F5B8568 84C32546 706C9567	(*First number is* $2F5B856884C32546706C9567_{16}$)
	`NUM2`	14DF4098 85B81095 A3BC1284	(*The second number is* $14DF409885B81095A3BC1284_{16}$)
Output:	`NUM1`	1A7C44CF FF0B14B0 CCB082E3	(*Difference is* $1A7C44CFFF0B14B0CCB082E3_{16}$)

That is,

$$\begin{array}{rr} & 2F5B856884C32546706C9567 \\ - & 14DF409885B81095A3BC1284 \\ \hline & 1A7C44CFFF0B14B0CCB082E3 \end{array}$$

12.2.2 Decimal Subtraction

Subtract one packed decimal (BCD) number from another. The length in bytes of both numbers is in the byte variable `LENGTH`. The numbers themselves are in the variables `NUM1` and `NUM2` respectively. Subtract the number contained in `NUM2` from the one contained in `NUM1`. Store the difference in `NUM1`.

Sample Problem:

Input:	LENGTH	4	(*Number of bytes in each number*)
	NUM1	36	(*The first number is* 367019857834_{10})
		70	
		19	
		85	
		78	
		34	
	NUM2	12	(*The second number is* 126634593269_{10})
		66	
		34	
		59	
		32	
		69	
Output:	NUM1	24	(*Difference is* 240385264565_{10})
		03	
		85	
		26	
		45	
		65	

That is,

$$\begin{array}{rr} & 367019857834 \\ - & 126634593269 \\ \hline & 240385264565 \end{array}$$

12.2.3 32-Bit by 32-Bit Multiply

Multiply the 32-bit value in the NUM1 variable by the value in the NUM2 variable. Use the MULU instruction and place the result in the 64-bit variable PROD1.

Sample Problem:

Input:	NUM1	0024	(*The first number is* $2468AC_{16}$)
		68AC	
	NUM2	0328	(*The second number is* 3281088_{10})
		1088	
Output:	PROD1	0000	
		72EC	(MULU *product is* $72ECB8C25B60_{16}$)
		B8C2	
		5B60	
	PROD2	0000	
		72EC	(*Shift product is* $72ECB8C25B60_{16}$)
		B8C2	
		5B60	

13 Tables and Lists

Tables and lists are two of the basic data structures used with all computers. We have already seen tables used to perform code conversions and arithmetic. Tables may also be used to identify or respond to commands and instructions, provide access to files or records, define the meaning of keys or switches, and choose among alternate programs. Lists are usually less structured than tables. Lists may record tasks that the processor must perform, messages or data that the processor must record, or conditions that have changed or should be monitored.

13.1 Program Examples

13.1.1 Add Entry to List

Program 13.1a: `insert.s` — *Examine a table for a match - store a new entry at the end if no match found*

```
1   *       examine a table for a match - store a new entry at
2   *       the end if no match found
3
4           TTL     Ch9Ex1
5           AREA    Program, CODE, READONLY
6           ENTRY
7
8   Main
9           LDR     R0, List                ;load the start address of the list
10          LDR     R1, NewItem             ;load the new item
11          LDR     R3, [R0]                ;copy the list counter
12          LDR     R2, [R0], #4            ;init counter and increment pointer
13          LDR     R4, [R0], #4
14  Loop
15          CMP     R1, R4                  ;does the item match the list?
16          BEQ     Done                    ;found it - finished
17          SUBS    R2, R2, #1              ;no - get the next item
18          LDR     R4, [R0], #4            ;get the next item
19          BNE     Loop                    ;and loop
20
21          SUB     R0, R0, #4              ;adjust the pointer
22          ADD     R3, R3, #1              ;increment the number of items
23          STR     R3, Start               ;and store it back
24          STR     R1, [R0]                ;store the new item at the end of the list
25
26  Done    SWI     &11
27
28          AREA    Data1, DATA
29  Start   DCD     &4                      ;length of list
30          DCD     &5376                   ;items
31          DCD     &7615
32          DCD     &138A
33          DCD     &21DC
34  Store   %       &20                     ;reserve 20 bytes of storage
```

```
35
36          AREA    Data2, DATA
37  NewItem DCD     &16FA
38  List    DCD     Start
39
40          END
```

Program 13.1b: `insert2.s` — *Examine a table for a match - store a new entry if no match found extends insert.s*

```
1   *       examine a table for a match - store a new entry if no match found
2   *       extends Ch9Ex1
3
4           TTL     Ch9Ex2
5           AREA    Program, CODE, READONLY
6           ENTRY
7
8   Main
9           LDR     R0, List                ;load the start address of the list
10          LDR     R1, NewItem             ;load the new item
11          LDR     R3, [R0]                ;copy the list counter
12          LDR     R2, [R0], #4            ;init counter and increment pointer
13          CMP     R3, #0                  ;it's an empty list
14          BEQ     Insert                  ;so store it
15          LDR     R4, [R0], #4            ;not empty - move to 1st item
16  Loop
17          CMP     R1, R4                  ;does the item match the list?
18          BEQ     Done                    ;found it - finished
19          SUBS    R2, R2, #1              ;no - get the next item
20          LDR     R4, [R0], #4            ;get the next item
21          BNE     Loop                    ;and loop
22
23          SUB     R0, R0, #4              ;adjust the pointer
24  Insert  ADD     R3, R3, #1              ;incr list count
25          STR     R3, Start               ;and store it
26          STR     R1, [R0]                ;store new item at the end
27
28  Done    SWI     &11                     ;all done
29
30          AREA    Data1, DATA
31  Start   DCD     &4                      ;length of list
32          DCD     &5376                   ;items
33          DCD     &7615
34          DCD     &138A
35          DCD     &21DC
36  Store   %       &20                     ;reserve 20 bytes of storage
37
38          AREA    Data2, DATA
39  NewItem DCD     &16FA
40  List    DCD     Start
41
42          END
```

13.1.2 Check an Ordered List

Program 13.2: `search.s` — *Examine an ordered table for a match*

```
1   *       examine an ordered table for a match
2
3           TTL     Ch9Ex3
4           AREA    Program, CODE, READONLY
5           ENTRY
6
7   Main
```

```
8           LDR     R0, =NewItem            ;load the address past the list
9           SUB     R0, R0, #4              ;adjust pointer to point at last element of list
10          LDR     R1, NewItem             ;load the item to test
11          LDR     R3, Start               ;init counter by reading index from list
12          CMP     R3, #0                  ;are there zero items
13          BEQ     Missing                 ;zero items in list - error condition
14          LDR     R4, [R0], #-4
15  Loop
16          CMP     R1, R4                  ;does the item match the list?
17          BEQ     Done                    ;found it - finished
18          BHI     Missing                 ;if the one to test is higher, it's not in the list
19          SUBS    R3, R3, #1              ;no - decr counter
20          LDR     R4, [R0], #-4           ;get the next item
21          BNE     Loop                    ;and loop
22                                          ;if we get to here, it's not there either
23  Missing MOV     R3, #0xFFFFFFFF         ;flag it as missing
24
25  Done    STR     R3, Index               ;store the index (either index or -1)
26          SWI     &11                     ;all done
27
28          AREA    Data1, DATA
29  Start   DCD     &4                      ;length of list
30          DCD     &0000138A               ;items
31          DCD     &000A21DC
32          DCD     &001F5376
33          DCD     &09018613
34
35          AREA    Data2, DATA
36  NewItem DCD     &001F5376
37  Index   DCW     0
38  List    DCD     Start
39
40          END
```

13.1.3 Remove an Element from a Queue

Program 13.3: `head.s` — *Remove the first element of a queue*

```
1   *       remove the first element of a queue
2
3           TTL     Ch9Ex4
4           AREA    Program, CODE, READONLY
5           ENTRY
6
7   Main
8           LDR     R0, Queue               ;load the head of the queue
9           STR     R1, Pointer             ;and save it in 'Pointer'
10          CMP     R0, #0                  ;is it NULL?
11          BEQ     Done                    ;if so, nothing to do
12
13          LDR     R1, [R0]                ;otherwise get the ptr to next
14          STR     R1, Queue               ;and make it the start of the queue
15
16  Done    SWI     &11
17
18          AREA    Data1, DATA
19  Queue   DCD     Item1                   ;pointer to the start of the queue
20  Pointer DCD     0                       ;space to save the pointer
21
22  DArea   %       20                      ;space for new entries
23
24  * each item consists of a pointer to the next item, and some data
25  Item1   DCD     Item2                   ;pointer
26          DCB     30, 20                  ;data
```

```
27
28  Item2   DCD     Item3                   ;pointer
29          DCB     30, 0xFF                ;data
30
31  Item3   DCD     0                       ;pointer (NULL)
32          DCB     30,&87,&65              ;data
33
34          END
```

13.1.4 Sort a List

Program 13.4: `sort.s` — *Sort a list of values – simple bubble sort*

```
1   *       sort a list of values - simple bubble sort
2
3           TTL     Ch9Ex5
4           AREA    Program, CODE, READONLY
5           ENTRY
6
7   Main
8           LDR     R6, List                ;pointer to start of list
9           MOV     R0, #0                  ;clear register
10          LDRB    R0, [R6]                ;get the length of list
11          MOV     R8, R6                  ;make a copy of start of list
12  Sort
13          ADD     R7, R6, R0              ;get address of last element
14          MOV     R1, #0                  ;zero flag for changes
15          ADD     R8, R8, #1              ;move 1 byte up the list each
16  Next                                    ;iteration
17          LDRB    R2, [R7], #-1           ;load the first byte
18          LDRB    R3, [R7]                ;and the second
19          CMP     R2, R3                  ;compare them
20          BCC     NoSwitch                ;branch if r2 less than r3
21
22          STRB    R2, [R7], #1            ;otherwise swap the bytes
23          STRB    R3, [R7]                ;like this
24          ADD     R1, R1, #1              ;flag that changes made
25          SUB     R7, R7, #1              ;decrement address to check
26  NoSwitch
27          CMP     R7, R8                  ;have we checked enough bytes?
28          BHI     Next                    ;if not, do inner loop
29          CMP     R1, #0                  ;did we mke changes
30          BNE     Sort                    ;if so check again - outer loop
31
32  Done    SWI     &11                     ;all done
33
34          AREA    Data1, DATA
35  Start   DCB     6
36          DCB     &2A, &5B, &60, &3F, &D1, &19
37
38          AREA    Data2, DATA
39  List    DCD     Start
40
41          END
```

13.1.5 Using an Ordered Jump Table

13.2 Problems

13.2.1 Remove Entry from List

Remove the value in the variable `ITEM` at a list if the value is present. The address of the list is in the `LIST` variable. The first entry in the list is the number (in words) of elements remaining in the list. Move entries below the one removed up one position and reduce the length of the list by one.

Sample Problems:

	Test A		Test B	
	Input	Output	Input	Output
`ITEM`	D010257		D0102596	
`LIST`	`Table`		`Table`	
Table	00000004	*No change*	00000004	*0003*
	2946C121	*since item*	C1212546	C1212546
	2054A346	*not in list*	D0102596	*3A64422B*
	05723A64		3A64422B	*6C20432E*
	12576C20		6C20432E	—

13.2.2 Add Entry to Ordered List

Insert the value in the variable `ITEM` into an ordered list if it is not already there. The address of the list is in the `LIST` variable. The first entry in the list is the list's length in words. The list itself consists of unsigned binary numbers in increasing order. Place the new entry in the correct position in the list, adjust the element below it down, and increase the length of the list by one.

Sample Problems

	Test A		Test B	
	Input	Output	Input	Output
`ITEM`	7A35B310		7A35B310	
`LIST`	*Table*		*Table*	
Table	00000004	*0005*	00000005	*No change*
	09250037	09250037	09250037	*since ITEM*
	29567322	29567322	29567322	*already in*
	A356A101	*7A35B310*	7A35B310	*list.*
	E235C203	*A356A101*	A356A101	
	—	*E235C203*	E235C203	

13.2.3 Add Element to Queue

Add the value in the variable `ITEM` to a queue. The address of the first element in the queue is in the variable `QUEUE`. Each element in the queue contains a structure of two items (*value* and *next*) where *next* is either the address of the next element in the queue or zero if there is no next element. The new element is placed at the end (tail) of the queue; the new element's address will be in the element that *was* at the end of the queue. The *next* entry of the new element will contain zero to indicate that it is now the end of the queue.

Sample Problem:

	Input		Output	
	Value	*Next*	*Value*	*Next*
`ITEM`	23854760		23854760	
`QUEUE`	00000001	*item1*	00000001	*item1*
item1	00000123	*item2*	00000123	*item2*
item2	00123456	00000000	00123456	***item3***
item3	—	—	**23854760**	***00000000***

13.2.4 4-Byte Sort

Sort a list of 4-byte entries into descending order. The first three bytes in each entry are an unsigned key with the first byte being the most significant. The fourth byte is additional information and should not be used to determine the sort order, but should be moved along with its key. The number of entries in the list is defined by the word variable `LENGTH`. The list itself begins at location `LIST`.

Sample Problem:

	Input		Output	
`LENGTH`	00000004			
`LIST`	414243 07	(“ABC”)	*4A4B4C 13*	(“JKL”)
	4A4B4C 13	(“JKL”)	*4A4B41 37*	(“JKA”)
	4A4B41 37	(“JKA”)	*444B41 3F*	(“DKA”)
	444B41 3F	(“DKA”)	*414243 07*	(“ABC”)

13.2.5 Using a Jump Table with a Key

Using the value in the variable `INDEX` as a key to a jump table (`TABLE`). Each entry in the jump table contains a 32-bit identifier followed by a 32-bit address to which the program should transfer control if the key is equal to that identifier.

Sample Problem:

```
INDEX   00000010

TABLE   00000001   Proc1
        00000010   Proc2
        0000001E   Proc3

Proc1   NOP
Proc2   NOP
Proc3   NOP
```

Control should be transferred to `Proc2`, the second entry in the table.

14 Subroutines

None of the examples that we have shown thus far is a typical program that would stand by itself. Most real programs perform a series of tasks, many of which may be used a number of times or be common to other programs.

The standard method of producing programs which can be used in this manner is to write subroutines that perform particular tasks. The resulting sequences of instructions can be written once, tested once, and then used repeatedly.

There are special instructions for transferring control to subroutines and restoring control to the main program. We often refer to the special instruction that transfers control to a subroutine as Call, Jump, or Brach to a Subroutine. The special instruction that restores control to the main program is usually called Return.

In the ARM the Branch-and-Link instruction (`BL`) is used to Branch to a Subroutine. This saves the current value of the program counter (PC or R*15*) in the Link Register (LR or R*14*) before placing the starting address of the subroutine in the program counter. The ARM does not have a standard Return from Subroutine instruction like other processors, rather the programmer should copy the value in the Link Register into the Program Counter in order to return to the instruction after the Branch-and-Link instruction. Thus, to return from a subroutine you should the instruction:

```
MOV    PC, LR
```

Should the subroutine wish to call another subroutine it will have to save the value of the Link Register before calling the nested subroutine.

14.1 Types of Subroutines

Sometimes a subroutine must have special characteristics.

Relocatable
: The code can be placed anywhere in memory. You can use such a subroutine easily, regardless of other programs or the arrangement of the memory. A relocating loader is necessary to place the program in memory properly; the loader will start the program after other programs and will add the starting address or relocation constant to all addresses in the program.

Position Independent
: The code does not require a relocating loader — all program addresses are expressed relative to the program counter's current value. Data addresses are held in-registers at all times. We will discuss the writing of position independent code later in this chapter.

Reentrant
: The subroutine can be interrupted and called by the interrupting program, giving the correct results for both the interrupting and interrupted programs. Reentrant subroutines are required for good for event based systems such as a multitasking operating system (Windows or Unix) and embedded real time environments. It is not difficult to make a subroutine

reentrant. The only requirement is that the subroutine uses just registers and the stack for its data storage, and the subroutine is self contained in that it does not use any value defined outside of the routine (global values).

Recursive
The subroutine can call itself. Such a subroutine clearly must also be reentrant.

14.2 Subroutine Documentation

Most programs consist of a main program and several subroutines. This is useful as you can use known pre-written routines when available and you can debug and test the other subroutines properly and remember their exact effects on registers and memory locations.

You should provide sufficient documentation such that users need not examine the subroutine's internal structure. Among necessary specifications are:

- A description of the purpose of the subroutine
- A list of input and output parameters
- Registers and memory locations used
- A sample case, perhaps including a sample calling sequence

The subroutine will be easy to use if you follow these guidelines.

14.3 Parameter Passing Techniques

In order to be really useful, a subroutine must be general. For example, a subroutine that can perform only a specialized task, such as looking for a particular letter in an input string of fixed length, will not be very useful. If, on the other hand, the subroutine can look for any letter, in strings of any length, it will be far more helpful.

In order to provide subroutines with this flexibility, it is necessary to provide them with the ability to receive various kinds of information. We call data or addresses that we provide the subroutine parameters. An important part of writing subroutines is providing for transferring the parameters to the subroutine. This process is called Parameter Passing.

There are three general approaches to passing parameters:

1. Place the parameters in registers.
2. Place the parameters in a block of memory.
3. Transfer the parameters and results on the hardware stack.

The registers often provide a fast, convenient way of passing parameters and returning results. The limitations of this method are that it cannot be expanded beyond the number of available registers; it often results in unforeseen side effects; and it lacks generality.

The trade-off here is between fast execution time and a more general approach. Such a trade-off is common in computer applications at all levels. General approaches are easy to learn and consistent; they can be automated through the use of macros. On the other hand, approaches that take advantage of the specific features of a particular task require less time and memory. The choice of one approach over the other depends on your application, but you should take the general approach (saving programming time and simplifying documentation and maintenance) unless time or memory constraints force you to do otherwise.

14.3.1 Passing Parameters In Registers

The first and simplest method of passing parameters to a subroutine is via the registers. After calling a subroutine, the calling program can load memory addresses, counters, and other data into registers. For example, suppose a subroutine operates on two data buffers of equal length. The subroutine might specify that the length of the two data buffers be in the register R*0* while the staring address of the two data buffer are in the registers R*1* and R*2*. The calling program would then call the subroutine as follows:

```
MOV   R0, #BufferLen   ; Length of Buffer in R0
LDR   R1, =BufferA     ; Buffer A beginning address in R1
LDR   R2, =BufferB     ; Buffer B beginning address in R2
BL    Subr             ; Call subroutine
```

Using this method of parameter passing, the subroutine can simply assume that the parameters are there. Results can also be returned in registers, or the addresses of locations for results can be passed as parameters via the registers. Of course, this technique is limited by the number of registers available.

Processor features such as register indirect addressing, indexed addressing, and the ability to use any register as a stack pointer allow far more powerful and general ways of passing parameters.

14.3.2 Passing Parameters In A Parameter Block

Parameters that are to be passed to a subroutine can also be placed into memory in a parameter block. The location of this parameter block can be passed to the subroutine via a register.

```
LDR   R0, =Params   ; R0 Points to Parameter Block
BL    Subr          ; Call the subroutine
```

If you place the parameter block immediately after the subroutine call the address of the parameter block is automatically place into the Link Register by the Branch and Link instruction. The subroutine must modify the return address in the Link Register in addition to fetching the parameters. Using this technique, our example would be modified as follows:

```
BL    Subr
DCD   BufferLen   ;Buffer Length
DCD   BufferA     ;Buffer A starting address
DCD   BufferB     ;Buffer B starting address
```

The subroutine saves' prior contents of CPU registers, then loads parameters and adjusts the return address as follows:

```
Subr   LDR   R0, [LR], #4   ; Read BuufferLen
       LDR   R1, [LR], #4   ; Read address of Buffer A
       LDR   R2, [LR], #4   ; Read address of Buffer B
                            ; LR points to next instruction
```

The addressing mode `[LR], #4` will read the value at the address pointed to by the Link Register and then move the register on by four bytes. Thus at the end of this sequence the value of LR has been updated to point to the next instruction after the parameter block.

This parameter passing technique has the advantage of being easy to read. It has, however, the disadvantage of requiring parameters to be *fixed* when the program is written. Passing the address of the parameter block in via a register allows the papa meters to be changed as the program is running.

14.3.3 Passing Parameters On The Stack

Another common method of passing parameters to a subroutine is to push the parameters onto the stack. Using this parameter passing technique, the subroutine call illustrated above would occur as follows:

```
MOV   R0, #BufferLen   ; Read Buffer Length
STR   R0, [SP, #-4]!   ; Save on the stack
LDR   R0, =BufferA     ; Read Address of Buffer A
STR   R0, [SP, #-4]!   ; Save on the stack
LDR   R0, =BufferA     ; Read Address of Buffer B
STR   R0, [SP, #-4]!   ; Save on the stack
BL    Subr
```

The subroutine must begin by loading parameters into CPU registers as follows:

```
Subr   STMIA   R12, {R0, R1, R2, R12, R14}   ; save working registers to stack
       LDR     R0, [R12, #0]                 ; Buffer Length in D0
       LDR     R1, [R12, #4]                 ; Buffer A starting address
       LDR     R2, [R12, #8]                 ; Buffer B starting address
       ...                                   ; Main function of subroutine
       LDMIA   R12, {R0, R1, R2, R12, R14}   ; Recover working registers
       MOV     PC, LR                        ; Return to caller
```

In this approach, all parameters are passed and results are returned on the stack.

The stack grows downward (toward lower addresses). This occurs because elements are pushed onto the stack using the pre-decrement address mode. The use of the pre-decrement mode causes the stack pointer to always contain the address of the last occupied location, rather than the next empty one as on some other microprocessors. This implies that you must initialise the stack pointer to a value higher than the largest address in the stack area.

When passing parameters on the stack, the programmer must implement this approach as follows:

1. Decrement the system stack pointer to make room for parameters on the system stack, and store them using offsets from the stack pointer, or simply push the parameters on the stack.
2. Access the parameters by means of offsets from the system stack pointer.
3. Store the results on the stack by means of offsets from the systems stack pointer.
4. Clean up the stack before or after returning from the subroutine, so that the parameters are removed and the results are handled appropriately.

14.4 Types Of Parameters

Regardless of our approach to passing parameters, we can specify the parameters in a variety of ways. For example, we can:

pass-by-value
: Where the actual values are placed in the parameter list. The name comes from the fact that it is only the value of the parameter that is passed into the subroutine rather than the parameter itself. This is the method used by most high level programming languages.

pass-by-reference
: The address of the parameters are placed in the parameter list. The subroutine can access the value directly rather than a copy of the parameter. This is much more dangerous as the subroutine can change a value you don't want it to.

pass-by-name
: Rather than passing either the value or a reference to the value a string containing the name

of the parameter is passed. This is used by very high level languages or scripting languages. This is very flexible but rather time consuming as we need to look up the value associated with the variable name every time we wish to access the variable.

14.5 Program Examples

Program 14.1a: `init1.s` — *Initiate a simple stack*

```
1   *       initiate a simple stack
2
3           TTL     Ch10Ex1
4           AREA    Program, CODE, READONLY
5           ENTRY
6
7   Main
8           LDR     R1, Value1              ;put some data into registers
9           LDR     R2, Value2
10          LDR     R3, Value3
11          LDR     R4, Value4
12
13          LDR     R7, =Data2              ;load the top of stack
14          STMFD   R7, {R1 - R4}           ;push the data onto the stack
15
16          SWI     &11                     ;all done
17
18          AREA    Stack1, DATA
19  Value1  DCD     0xFFFF
20  Value2  DCD     0xDDDD
21  Value3  DCD     0xAAAA
22  Value4  DCD     0x3333
23
24          AREA    Data2, DATA
25  Stack   %       40                      ;reserve 40 bytes of memory for the stack
26  StackEnd
27          DCD     0
28
29          END
```

Program 14.1b: `init2.s` — *Initiate a simple stack*

```
1   *       initiate a simple stack
2
3           TTL     Ch10Ex2
4           AREA    Program, CODE, READONLY
5           ENTRY
6
7   Main
8           LDR     R1, Value1              ;put some data into registers
9           LDR     R2, Value2
10          LDR     R3, Value3
11          LDR     R4, Value4
12
13          LDR     R7, =Data2
14          STMDB   R7, {R1 - R4}
15
16          SWI     &11                     ;all done
17
18          AREA    Stack1, DATA
19  Value1  DCD     0xFFFF
20  Value2  DCD     0xDDDD
21  Value3  DCD     0xAAAA
22  Value4  DCD     0x3333
23
```

```
24          AREA    Data2, DATA
25  Stack   %       40                      ;reserve 40 bytes of memory for the stack
26  StackEnd
27          DCD     0
28
29          END
```

Program 14.1c: `init3.s` — *Initiate a simple stack*

```
1   *       initiate a simple stack
2
3           TTL     Ch10Ex3
4           AREA    Program, CODE, READONLY
5           ENTRY
6
7   StackStart      EQU     0x9000
8   Main
9           LDR     R1, Value1              ;put some data into registers
10          LDR     R2, Value2
11          LDR     R3, Value3
12          LDR     R4, Value4
13
14          LDR     R7, =StackStart         ;Top of stack = 9000
15          STMDB   R7, {R1 - R4}           ;push R1-R4 onto stack
16
17          SWI     &11                     ;all done
18
19          AREA    Data1, DATA
20  Value1  DCD     0xFFFF                  ;some data to put on stack
21  Value2  DCD     0xDDDD
22  Value3  DCD     0xAAAA
23  Value4  DCD     0x3333
24
25          AREA    Data2, DATA
26          ^       StackStart              ;reserve 40 bytes of memory for the stack
27  Stack1  DCD     0
28
29          END
```

Program 14.1d: `init3a.s` — *Initiate a simple stack*

```
1   *       initiate a simple stack
2
3           TTL     Ch10Ex4
4           AREA    Program, CODE, READONLY
5           ENTRY
6
7   StackStart      EQU     0x9000
8   start
9           LDR     R1, Value1              ;put some data into registers
10          LDR     R2, Value2
11          LDR     R3, Value3
12          LDR     R4, Value4
13
14          LDR     R7, =StackStart         ;Top of stack = 9000
15          STMDB   R7, {R1 - R4}           ;push R1-R4 onto stack
16
17          SWI     &11                     ;all done
18
19          AREA    Data1, DATA
20  Value1  DCD     0xFFFF
21  Value2  DCD     0xDDDD
22  Value3  DCD     0xAAAA
23  Value4  DCD     0x3333
24
25          AREA    Data2, DATA
```

```
26          ^       StackStart              ;reserve 40 bytes of memory for the stack
27  Stack1  DCD     0
28
29          END
```

Program 14.1e: `byreg.s` — *A simple subroutine example program passes a variable to the routine in a register*

```
1   *       a simple subroutine example
2   *       program passes a variable to the routine in a register
3
4           TTL     Ch10Ex4
5           AREA    Program, CODE, READONLY
6           ENTRY
7
8   StackStart      EQU     0x9000
9   Main
10          LDRB    R0, HDigit              ;variable stored to register
11          BL      Hexdigit                ;branch/link
12          STRB    R0, AChar               ;store the result of the subroutine
13          SWI     &0                      ;output to console
14          SWI     &11                     ;all done
15
16  *       =========================
17  *       Hexdigit subroutine
18  *       =========================
19
20  *       Purpose
21  *       Hexdigit subroutine converts a Hex digit to an ASCII character
22  *
23  *       Initial Condition
24  *       R0 contains a value in the range 00 ... 0F
25  *
26  *       Final Condition
27  *       R0 contains ASCII character in the range '0' ... '9' or 'A' ... 'F'
28  *
29  *       Registers changed
30  *       R0 only
31  *
32  *       Sample case
33  *       Initial condition       R0 = 6
34  *       Final condition         R0 = 36 ('6')
35
36  Hexdigit
37          CMP     R0, #0xA                ;is it > 9
38          BLE     Addz                    ;if not skip the next
39          ADD     R0, R0, #"A" - "0" - 0xA        ;adjust for A .. F
40
41  Addz
42          ADD     R0, R0, #"0"            ;convert to ASCII
43          MOV     PC, LR                  ;return from subroutine
44
45          AREA    Data1, DATA
46  HDigit  DCB     6                       ;digit to convert
47  AChar   DCB     0                       ;storage for ASCII character
48
49          END
```

Program 14.1f: `bystack.s` — *A more complex subroutine example program passes variables to the routine using the stack*

```
1   *       a more complex subroutine example
2   *       program passes variables to the routine using the stack
3
4           TTL     Ch10Ex5
5           AREA    Program, CODE, READONLY
```

```
6           ENTRY
7
8   StackStart      EQU     0x9000          ;declare where top of stack will be
9   Mask            EQU     0x0000000F      ;bit mask for masking out lower nibble
10
11  Main
12          LDR     R7, =StackStart         ;Top of stack = 9000
13          LDR     R0, Number              ;Load number to register
14          LDR     R1, =String             ;load address of string
15          STR     R1, [R7], #-4           ;and store it
16          STR     R0, [R7], #-4           ;and store number to stack
17          BL      Binhex                  ;branch/link
18          SWI     &11                     ;all done
19
20  *       =========================
21  *       Binhex subroutine
22  *       =========================
23
24  *       Purpose
25  *       Binhex subroutine converts a 16 bit value to an ASCII string
26  *
27  *       Initial Condition
28  *       First parameter on the stack is the value
29  *       Second parameter is the address of the string
30  *
31  *       Final Condition
32  *       the HEX string occupies 4 bytes beginning with
33  *       the address passed as the second parameter
34  *
35  *       Registers changed
36  *       No registers are affected
37  *
38  *       Sample case
39  *       Initial condition       top of stack :  4CD0
40  *                               Address of string
41  *       Final condition         The string '4''C''D''0' in ASCII
42  *                               occupies memory
43
44  Binhex
45          MOV     R8, R7                  ;save stack pointer for later
46          STMDA   R7, {R0-R6,R14}         ;push contents of R0 to R6, and LR onto the stack
47          MOV     R1, #4                  ;init counter
48          ADD     R7, R7, #4              ;adjust pointer
49          LDR     R2, [R7], #4            ;get the number
50          LDR     R4, [R7]                ;get the address of the string
51          ADD     R4, R4, #4              ;move past the end of where the string is to be stored
52
53  Loop
54          MOV     R0, R2                  ;copy the number
55          AND     R0, R0, #Mask           ;get the low nibble
56          BL      Hexdigit                ;convert to ASCII
57          STRB    R0, [R4], #-1           ;store it
58          MOV     R2, R2, LSR #4          ;shift to next nibble
59          SUBS    R1, R1, #1              ;decr counter
60          BNE     Loop                    ;loop while still elements left
61
62          LDMDA   R8, {R0-R6,R14}         ;restore the registers
63          MOV     PC, LR                  ;return from subroutine
64
65  *       =========================
66  *       Hexdigit subroutine
67  *       =========================
68
69  *       Purpose
70  *       Hexdigit subroutine converts a Hex digit to an ASCII character
71  *
72  *       Initial Condition
```

```
73  *       R0 contains a value in the range 00 ... 0F
74  *
75  *       Final Condition
76  *       R0 contains ASCII character in the range '0' ... '9' or 'A' ... 'F'
77  *
78  *       Registers changed
79  *       R0 only
80  *
81  *       Sample case
82  *       Initial condition       R0 = 6
83  *       Final condition         R0 = 36 ('6')
84
85  Hexdigit
86          CMP     R0, #0xA                ;is the number 0 ... 9?
87          BLE     Addz                    ;if so, branch
88          ADD     R0, R0, #"A" - "0" - 0xA        ;adjust for A ... F
89
90  Addz
91          ADD     R0, R0, #"0"            ;change to ASCII
92          MOV     PC, LR                  ;return from subroutine
93
94          AREA    Data1, DATA
95  Number  DCD     &4CD0                   ;number to convert
96  String  DCB     4, 0                    ;counted string for result
97
98          END
```

Program 14.1g: `add64.s` — *A 64 bit addition subroutine*

```
1   *       a 64 bit addition subroutine
2
3           TTL     Ch10Ex6
4           AREA    Program, CODE, READONLY
5           ENTRY
6
7   Main
8           BL      Add64                   ;branch/link
9           DCD     Value1                  ;address of parameter 1
10          DCD     Value2                  ;address of parameter 2
11
12          SWI     &11                     ;all done
13
14
15  *       =========================
16  *       Add64 subroutine
17  *       =========================
18
19  *       Purpose
20  *       Add two 64 bit values
21  *
22  *       Initial Condition
23  *       The two parameter values are passed immediately
24  *       following the subroutine call
25  *
26  *       Final Condition
27  *       The sum of the two values is returned in R0 and R1
28  *
29  *       Registers changed
30  *       R0 and R1 only
31  *
32  *       Sample case
33  *       Initial condition
34  *       para 1 =   = &0420147AEB529CB8
35  *       para 2 =   = &3020EB8520473118
36  *
37  *       Final condition
38  *       R0 = &34410000
```

```
39  *       R1 = &0B99CDD0
40
41  Add64
42          STMIA   R12, {R2, R3, R14}      ;save registers to stack
43          MOV     R7, R12                 ;copy stack pointer
44          SUB     R7, R7, #4              ;adjust to point at LSB of 2nd value
45          LDR     R3, [R7], #-4           ;load successive bytes
46          LDR     R2, [R7], #-4
47          LDR     R1, [R7], #-4
48          LDR     R0, [R7], #-4
49
50          ADDS    R1, R1, R3              ;add LS bytes & set carry flag
51          BCC     Next                    ;branch if carry bit not set
52          ADD     R0, R0, #1              ;otherwise add the carry
53  Next
54          ADD     R0, R0, R2              ;add MS bytes
55          LDMIA   R12, {R2, R3, R14}      ;pop from stack
56          MOV     PC, LR                  ;and return
57
58          AREA    Data1, DATA
59  Value1  DCD     &0420147A, &EB529CB8    ;number1 to add
60  Value2  DCD     &3020EB85, &20473118    ;number2 to add
61          END
```

Program 14.1h: `factorial.s` — *A subroutine to find the factorial of a number*

```
1   *       a subroutine to find the factorial of a number
2
3           TTL     Ch10Ex6
4           AREA    Program, CODE, READONLY
5           ENTRY
6
7   Main
8           LDR     R0, Number              ;get number
9           BL      Factor                  ;branch/link
10          STR     R0, FNum                ;store the factorial
11
12          SWI     &11                     ;all done
13
14
15  *       =========================
16  *       Factor subroutine
17  *       =========================
18
19  *       Purpose
20  *       Recursively find the factorial of a number
21  *
22  *       Initial Condition
23  *       R0 contains the number to factorial
24  *
25  *       Final Condition
26  *       R0 = factorial of number
27  *
28  *       Registers changed
29  *       R0 and R1 only
30  *
31  *       Sample case
32  *       Initial condition
33  *       Number = 5
34  *
35  *       Final condition
36  *       FNum = 120 = 0x78
37
38  Factor
39          STR     R0, [R12], #4           ;push to stack
40          STR     R14, [R12], #4          ;push the return address
41          SUBS    R0, R0, #1              ;subtract 1 from number
```

```
42          BNE     F_Cont                  ;not finished
43
44          MOV     R0, #1                  ;Factorial == 1
45          SUB     R12, R12, #4            ;adjust stack pointer
46          B       Return                  ;done
47
48  F_Cont
49          BL      Factor                  ;if not done, call again
50
51  Return
52          LDR     R14, [R12], #-4         ;return address
53          LDR     R1, [R12], #-4          ;load to R1 (can't do MUL R0, R0, xxx)
54          MUL     R0, R1, R0              ;multiply the result
55          MOV     PC, LR                  ;and return
56
57          AREA    Data1, DATA
58  Number  DCD     5                       ;number
59  FNum    DCD     0                       ;factorial
60          END
```

14.6 Problems

Write both a calling program for the sample problem and at least one properly documented subroutine for each problem.

14.6.1 ASCII Hex to Binary

Write a subroutine to convert the least significant eight bits in register R0 from the ASCII representation of a hexadecimal digit to the 4-bit binary representation of the digit. Place the result back into R0.

Sample Problems:

		Test A	*Test B*
Input:	R0	43 'C'	36 '6'
Output:	R0	0C	06

14.6.2 ASCII Hex String to Binary Word

Write a subroutine that takes the address of a string of eight ASCII characters in R0. It should convert the hexadecimal string into a 32-bit binary number, which it return is R0.

Sample Problem:

Input:	R0	*String*
	STRING	42 'B'
		32 '2'
		46 'F'
		30 '0'
Output:	R0	0000B2F0

14.6.3 Test for Alphabetic Character

Write a subroutine that checks the character in register R0 to see if it is alphabetic (upper- or lower-case). It should set the Zero flag if the character is alphabetic, and reset the flag if it is not.

Sample Problems:

		Test A	*Test B*	*Test C*
Input:	R0	47 'G'	36 '6'	6A 'j'
Output:	Z	FF	00	FF

14.6.4 Scan to Next Non-alphabetic

Write a subroutien that takes the address of the start of a text string in register R1 and returns the address of the first non-alphabetic character in the string in register R1. You should consider using the *isalpha* subroutine you have just define.

Sample Problems:

		Test A	*B*
Input:	R1	*String*	6100
	String	43 'C'	32 '2'
		61 'a'	50 'P'
		74 't'	49 'I'
		0D CR	0D CR
Output:	R1	*String* + 4	*String* + 0
		(CR)	(2)

14.6.5 Check Even Parity

Write a subroutine that takes the address of a counted string in the register R0. It should check for an even number of set bits in each character of the string. If all the bytes have an even parity then it should set the Z-flag, if one or more bytes have an odd parity it should clear the Z-flag.

Sample Problems:

		Test A	*Test B*
Input:	R0	*String*	*String*
	String	03	03
		47	47
		AF	AF
		18	19
Output:	Z	00	FF

Note that 19_{16} is 0001 1001_2 which has three 1 bits and is thus has an odd parity.

14.6.6 Check the Checksum of a String

Write a subroutine to calculate the 8-bit checksum of the counted string pointed to by the register R0 and compares the calculated checksum with the 8-bit checksum at the end of the string. It should set the Z-flag if the checksums are equal, and reset the flag if they are not.

Sample Problems:

		Test A		Test B	
Input:	R0	*String*		*String*	
	String	03	(Length)	03	(Length)
		41	('A')	61	('a')
		42	('B')	62	('b')
		43	('C')	63	('c')
		C6	(Checksum)	C6	(*Checksum should be 26*)
Output:	Z	Set		Clear	

14.6.7 Compare Two Counted Strings

Write a subroutine to compare two ASCII strings. The first byte in each string is its length. Return the result in the condition codes; i.e., the N-flag will be set if the first string is lexically less than (prior to) the second, the Z-flag will be set if the strings are equal, no flags are set if the second is prior to the first. Note that "ABCD" is lexically greater than "ABC".

A ARM Instruction Definitions

This appendix describes every ARM instruction, in terms of:

Syntax This is a description of the way the instruction should be written in the assembler. A number of shorthand descriptions are used. The most common of these are:

⟨*cc*⟩	Condition Codes	(see section 4.1.2 on page 42)
⟨*op1*⟩	Data Movement Addressing Modes	(see section 5.1 on page 51)
⟨*op2*⟩	Memory Addressing Modes	(see section 5.2 on page 54)
⟨*S*⟩	Set Flags bit	(see section 4.1.1 on page 41)

Additional notations will be described as they are introduced.

Operation A Register Transfer Language (RTL) / pseudo-code description of what the instruction does. This will use the same shorthand notations used for the Syntax. For details of the register transfer language, see section 3.5 on page 30.

Description Written description of what the instruction does. This will interpret the formal description given in the operation part.

Exceptions This gives details of which exceptions can occur during the instruction. Prefetch Abort is not listed because it can occur for any instruction.

Usage Suggestions and other information relating to how an instruction can be used effectively.

Condition Codes Indicates what happens to the CPU Condition Code Flags if the set flags option where to be set.

Notes Contain any additional explanation that we can not fit into the previous categories.

Appendix B provides a summary of the more common instructions in a more compact manner, using the operation section only.

ADC **Add with Carry**

Syntax `ADC`⟨*cc*⟩⟨*S*⟩ R*d*, R*n*, ⟨*op1*⟩

Operation

⟨*cc*⟩: R*d* ← R*n* + ⟨*op1*⟩ + C
⟨*cc*⟩⟨*S*⟩: CPSR ← ALU(flags)

Description The `ADC` (Add with Carry) instruction adds the value of ⟨*op1*⟩ and the Carry flag to the value of R*n* and stores the result in R*d*. The condition code flags are optionally updated, based on the result.

Usage `ADC` is used to synthesize multi-word addition. If register pairs R*0*, R*1* and R*2*, R*3* hold 64-bit values (where R*0* and R*2* hold the least significant words) the following instructions leave the 64-bit sum in R*4*, R*5*:

```
ADDS  R4,R0,R2
ADC   R5,R1,R3
```

If the second instruction is changed from:

```
ADC   R5,R1,R3
```

to:

```
ADCS  R5,R1,R3
```

the resulting values of the flags indicate:

N The 64-bit addition produced a negative result.

C An unsigned overflow occurred.

V A signed overflow occurred.

Z The most significant 32 bits are all zero.

The following instruction produces a single-bit Rotate Left with Extend operation (33-bit rotate through the Carry flag) on R*0*:

```
ADCS  R0,R0,R0
```

See *Data-processing operands - Rotate right with extend* for information on how to perform a similar rotation to the right.

Condition Codes

The N and Z flags are set according to the result of the addition, and the C and V flags are set according to whether the addition generated a carry (unsigned overflow) and a signed overflow, respectively.

Notes If the destination register (R*d*) is the program counter (PC or R*15*), then the current status register (CPSR) is restored from the saved status register (SPSR). This form of the instruction is unpredictable if executed in User mode or System mode, because these modes do not have an SPSR.

ADD **Add**

Syntax ADD⟨*cc*⟩⟨*S*⟩ R*d*, R*n*, ⟨*op1*⟩

Operation

⟨*cc*⟩: R*d* ← R*n* + ⟨*op1*⟩
⟨*cc*⟩⟨*S*⟩: CPSR ← ALU(flags)

Description Adds the value of ⟨*op1*⟩ to the value of register R*n*, and stores the result in the destination register R*d*. The condition code flags are optionally updated, based on the result.

Usage The ADD instruction is used to add two values together to produce a third.

To increment a register value in R*x* use:

```
ADD   Rx, Rx, #1
```

Constant multiplication of R*x* by $2^n + 1$ into R*d* can be performed with:

```
ADD   Rd, Rx, Rx, LSL #n
```

To form a PC-relative address use:

```
ADD   Rs, PC, #offset
```

where the ⟨*offset*⟩ must be the difference between the required address and the address held in the PC, where the PC is the address of the `ADD` instruction itself plus 8 bytes.

Condition Codes
The N and Z flags are set according to the result of the addition, and the C and V flags are set according to whether the addition generated a carry (unsigned overflow) and a signed overflow, respectively.

Notes If the destination register (R*d*) is the program counter (PC or R*15*), then the current status register (CPSR) is restored from the saved status register (SPSR). This form of the instruction is unpredictable if executed in User mode or System mode, because these modes do not have an SPSR.

AND **Bitwise AND**

Syntax `AND`⟨*cc*⟩⟨*S*⟩ R*d*, R*n*, ⟨*op1*⟩

Operation
⟨*cc*⟩: R*d* ← R*n* ∧ ⟨*op1*⟩
⟨*cc*⟩⟨*S*⟩: CPSR ← ALU(flags)

Description The `AND` instruction performs a bitwise AND of the value of register R*n* with the value of ⟨*op1*⟩, and stores the result in the destination register R*d*. The condition code flags are optionally updated, based on the result.

Usage `AND` is most useful for extracting a field from a register, by *and*ing the register with a mask value that has *1*s in the field to be extracted, and *0*s elsewhere.

Condition Codes
The N and Z flags are set according to the result of the operation, and the C flag is set to the carry output generated by ⟨*op1*⟩ (see 5.1 on page 51), the V flag is unaffected.

Notes If the destination register (R*d*) is the program counter (PC or R*15*), then the current status register (CPSR) is restored from the saved status register (SPSR). This form of the instruction is unpredictable if executed in User mode or System mode, because these modes do not have an SPSR.

B, BL **Branch, Branch and Link**

Syntax `B`⟨*L*⟩⟨*cc*⟩ ⟨*offset*⟩

Operation
⟨*cc*⟩⟨*L*⟩: LR ← PC + 8
⟨*cc*⟩: PC ← PC + ⟨*offset*⟩

Description The `B` (Branch) and `BL` (Branch and Link) instructions cause a branch to a target address, and provide both conditional and unconditional changes to program flow.

The `BL` (Branch and Link) instruction stores a return address in the link register (LR or R*14*).

The ⟨*offset*⟩ specifies the target address of the branch. The address of the next instruction is calculated by adding the offset to the program counter (PC) which contains the address of the branch instruction plus 8.

The branch instructions can specify a branch of approximately ±32MB.

Usage The BL instruction is used to perform a subroutine call. The return from subroutine is achieved by copying the LR to the PC. Typically, this is done by one of the following methods:

- Executing a `MOV PC,R14` instruction.
- Storing a group of registers and R14 to the stack on subroutine entry, using an instruction of the form:

  `STMFD R13!,{`⟨*registers*⟩`,R14}`

  and then restoring the register values and returning with an instruction of the form:

  `LDMFD R13!,{`⟨*registers*⟩`,PC}`

Condition Codes The condition codes are not effected by this instruction.

Notes Branching backwards past location zero and forwards over the end of the 32-bit address space is unpredictable.

BIC **Bit Clear**

Syntax `BIC`⟨*cc*⟩⟨*S*⟩ R*d*, R*n*, ⟨*op1*⟩

Operation

$$\begin{aligned} \langle cc\rangle: \quad & Rd \leftarrow Rn \wedge \overline{\langle op1\rangle} \\ \langle cc\rangle\langle S\rangle: \ & \mathsf{CPSR} \leftarrow \mathsf{ALU(flags)} \end{aligned}$$

Description The `BIC` (Bit Clear) instruction performs a bitwise AND of the value of register R*n* with the complement of the value of ⟨*op1*⟩, and stores the result in the destination register R*d*. The condition code flags are optionally updated, based on the result.

Usage The instruction can be used to clear selected bits in a register. For each bit, `BIC` with *1* clears the bit, and `BIC` with *0* leaves it unchanged.

Condition Codes The N and Z flags are set according to the result of the operation, and the C flag is set to the carry output generated by ⟨*op1*⟩ (see 5.1 on page 51), the V flag is unaffected.

Notes If the destination register (R*d*) is the program counter (PC or R*15*), then the current status register (CPSR) is restored from the saved status register (SPSR). This form of the instruction is unpredictable if executed in User mode or System mode, because these modes do not have an SPSR.

CMN **Compare Negative**

Syntax `CMN`⟨*cc*⟩ R*n*, ⟨*op1*⟩

Operation

$$\begin{aligned} \langle cc\rangle: \ & \mathsf{ALU(0)} \leftarrow Rn + \langle op1\rangle \\ \langle cc\rangle: \ & \mathsf{CSPR} \leftarrow \mathsf{ALU(Flags)} \end{aligned}$$

Description The `CMN` (Compare Negative) instruction compares a register value with the negative of another arithmetic value. The condition flags are updated, based on the result of adding the second arithmetic value to the register value, so that subsequent instructions can be conditionally executed.

Usage The instruction performs a comparison by adding the value of ⟨*op1*⟩ to the value of register R*n*, and updates the condition code flags (based on the result). This is

almost equivalent to subtracting the negative of the second operand from the first operand, and setting the flags on the result. The difference is that the flag values generated can differ when the second operand is 0 or 0x80000000.

For example, this instruction always leaves the C flag set:

```
CMP    Rn, #0
```

while this instruction always leaves the C flag clear:

```
CMN    Rn, #0
```

CMP **Compare**

Syntax `CMP`⟨*cc*⟩ R*n*, ⟨*op1*⟩

Operation
⟨*cc*⟩: ALU(0) ← R*n* − ⟨*op1*⟩
⟨*cc*⟩: CSPR ← ALU(Flags)

Description The `CMP` (Compare) instruction compares a register value with another arithmetic value. The condition flags are updated, based on the result of subtracting ⟨*op1*⟩ from R*n*, so that subsequent instructions can be conditionally executed.

Condition Codes
The N and Z flags are set according to the result of the subtraction, and the C and V flags are set according to whether the subtraction generated a borrow (unsinged underflow) and a signed overflow, respectively.

EOR **Exclusive OR**

Syntax `EOR`⟨*cc*⟩⟨*S*⟩ R*d*, R*n*, ⟨*op1*⟩

Operation
⟨*cc*⟩: R*d* ← R*n* ⊕ ⟨*op1*⟩
⟨*cc*⟩⟨*S*⟩: CPSR ← ALU(Flags)

Description The `EOR` (Exclusive OR) instruction performs a bitwise Exclusive-OR of the value of register R*n* with the value of ⟨*op1*⟩, and stores the result in the destination register R*d*. The condition code flags are optionally updated, based on the result.

Usage `EOR` can be used to invert selected bits in a register. For each bit, a *1* inverts that bit, and a *0* leaves it unchanged.

Condition Codes
The N and Z flags are set according to the result of the operation, and the C flag is set to the carry output bit generated by the shifter. The V flag is unaffected.

Notes If the destination register (R*d*) is the program counter (PC or R*15*), then the current status register (CPSR) is restored from the saved status register (SPSR). This form of the instruction is unpredictable if executed in User mode or System mode, because these modes do not have an SPSR.

LDM **Load Multiple**

Syntax `LDM`⟨*cc*⟩⟨*mode*⟩ R*n*⟨*!*⟩, {⟨*registers*⟩}

where ⟨*mode*⟩ is one of:

`IB`	Increment Before	`IA`	Increment After
`DB`	Decrement Before	`DA`	Decrement After

Operation

```
if ⟨cc⟩
   IA: MAR ← Rn
   IB: MAR ← Rn + 4
   DA: MAR ← Rn − (#⟨registers⟩ × 4) + 4
   DB: MAR ← Rn − (#⟨registers⟩ × 4)

          for each register Ri in ⟨registers⟩
             IB: MAR ← MAR + 4
             DB: MAR ← MAR − 4
                 Ri ← M(MAR)
             IA: MAR ← MAR + 4
             DA: MAR ← MAR − 4

   ⟨!⟩:   Rn ← MAR
end if
```

Description The `LDM` (Load Multiple) instruction is useful for block loads, stack operations and procedure exit sequences. It loads a subset, or possibly all, of the general-purpose registers from sequential memory locations.

The register R*n* points to the memory local to load the values from. Each of the registers listed in ⟨*registers*⟩ is loaded in turn, reading each value from the next memory address as directed by ⟨*mode*⟩.

The base register writeback option (⟨*!*⟩) causes the base register to be modified to hold the address of the final valued loaded.

The register are loaded in sequence, the lowest-numbered register from the lowest memory address, through to the highest-numbered register from the highest memory address.

If the PC (R*15*) is specified in the register list, the instruction causes a branch to the address loaded into the PC.

Exceptions **Data Abort** — This exception is generated if the address is not word aligned.

Condition Codes
The condition codes are not effected by this instruction.

Notes The base register R*n* should not be specified in ⟨*registers*⟩ when the base register writeback (⟨*!*⟩) is used.

LDR — Load Register

Syntax `LDR`⟨*cc*⟩ R*d*, ⟨*op2*⟩

Operation ⟨*cc*⟩: R*d* ← M(⟨*op2*⟩)

Description The `LDR` (Load Register) instruction loads a word from the memory address calculated by ⟨*op2*⟩ and writes it to register R*d*.

If the PC is specified as register R*d*, the instruction loads a data word which it treats as an address, then branches to that address.

Exceptions **Data Abort** — This exception is generated if the memory address produced by ⟨*op2*⟩ is not word aligned.

Usage Using the PC as the base register allows PC-relative addressing, which facilitates position-independent code. Combined with a suitable addressing mode, `LDR` allows 32-bit memory data to be loaded into a general-purpose register where its value

can be manipulated. If the destination register is the PC, this instruction loads a 32-bit address from memory and branches to that address.

To synthesise a Branch with Link, precede the `LDR` instruction with `MOV LR, PC`.

Condition Codes
The condition codes are not effected by this instruction.

Notes If ⟨*op2*⟩ specifies an address that is not word-aligned, the instruction will load a 32-bit value, but the value will be unpredictable. The `LDRB` (load byte) instruction should be used.

If ⟨*op2*⟩ specifies base register writeback (`!`), and the same register is specified for R*d* and R*n*, the results are unpredictable.

If the PC (R*15*) is specified for R*d*, the value must be word aligned otherwise the result is unpredictable.

LDRB **Load Register Byte**

Syntax `LDR`⟨*cc*⟩`B` R*d*, ⟨*op2*⟩

Operation
⟨*cc*⟩: R*d*(7:0) ← M(⟨*op2*⟩)
⟨*cc*⟩: R*d*(31:8) ← 0

Description The `LDRB` (Load Register Byte) instruction loads a byte from the memory address calculated by ⟨*op2*⟩, zero-extends the byte to a 32-bit word, and writes the word to register R*d*.

Exceptions **Data Abort** — This exception is generated if the memory address produced by ⟨*op2*⟩ is not word aligned.

Usage `LDRB` allows 8-bit memory data to be loaded into a general-purpose register where it can be manipulated.

Using the PC as the base register allows PC-relative addressing, to facilitate position-independent code.

Condition Codes
The condition codes are not effected by this instruction.

Notes If the PC (R*15*) is specified for R*d*, the result is unpredictable.

If ⟨*op2*⟩ specifies base register writeback (`!`), and the same register is specified for R*d* and R*n*, the results are unpredictable.

MLA **Multiply Accumulate**

Syntax `MLA`⟨*cc*⟩⟨*S*⟩ R*d*, R*m*, R*s*, R*n*

Operation
⟨*cc*⟩: ALU ← (R*m* × R*s*)(0:31)
⟨*cc*⟩: R*d* ← ALU + R*n*
⟨*cc*⟩⟨*S*⟩: CPSR ← ALU(flags)

Description The `MLA` (Multiply Accumulate) multiplies signed or unsigned operands to produce a 32-bit result, which is then added to a third operand, and written to the destination register. The condition code flags are optionally updated, based on the result.

As it only produces the lower 32-bits of the 64-bit product, it gives the same answer for multiplication of both signed and unsigned numbers. That is, it does not take the sign into account.

Condition Codes

The N and Z flags are set according to the result of the operation. The V flag is unaffected. Unfortunately the C flag will have a random value, and should not be used after this instruction.

Notes Specifying the same register for R*d* and R*m* has unpredictable results.

MOV **Move**

Syntax `MOV`⟨*cc*⟩⟨*S*⟩ R*d*, ⟨*op1*⟩

Operation

⟨*cc*⟩: R*d* ← ⟨*op1*⟩
⟨*cc*⟩⟨*S*⟩: CPSR ← ALU(Flags)

Description The `MOV` (Move) instruction moves the value of ⟨*op1*⟩ to the destination register R*d*. The condition code flags are optionally updated, based on the result.

Usage `MOV` is used to:

- Move a value from one register to another.
- Put a constant value into a register.
- Perform a shift without any other arithmetic or logical operation. A left shift by n can be used to multiply by 2^n.
- When the PC is the destination of the instruction, a branch occurs. The instruction:

  ```
  MOV  PC, LR
  ```

  can therefore be used to return from a subroutine (see instructions *B*, and *BL* on page 139 and chapter 14).

Condition Codes

The N and Z flags are set according to the value moved (post-shift if a shift is specified), and the C flag is set to the carry output bit generated by the shifter (see 5.1 on page 51). The V flag is unaffected.

Notes If the destination register (R*d*) is the program counter (PC or R*15*), then the current status register (CPSR) is restored from the saved status register (SPSR). This form of the instruction is unpredictable if executed in User mode or System mode, because these modes do not have an SPSR.

MRS **Move to Register from Status**

Syntax

`MRS`⟨*cc*⟩ R*d* , `CPSR`
`MRS`⟨*cc*⟩ R*d* , `SPSR`

Operation

⟨*cc*⟩⟨*CPSR*⟩: R*d* ← CPSR
⟨*cc*⟩⟨*SPSR*⟩: R*d* ← SPSR

Description The `MRS` (Move to Register from Status) instruction moves the value of the status register, either the current status (CPSR) or the saved status (SPSR) into a destination register R*d*. The value can then be examined, or manipulated, with normal data-processing instructions.

Usage This is commonly used for three purposes:

- As part of a read/modify/write sequence for updating a status (see `MSR` below for a discussion).
- When an exception occurs and there is a possibility of a nested exception of the same type occurring, the saved process status (SPSR) of the exception mode is in danger of being corrupted. The value can be copied out and saved before the nested exception can occur, and later restored in preparation for the exception return.
- When swapping process, the current state of the process being swapped out needs to be saved, including the current flags, and processor mode. Similarly the state of the process being swapped in needs to be restored.

Condition Codes
The condition codes are not effected by this instruction, just copied into the destination register.

Notes The user mode does not have a saved process status register (SPSR) so attempting to access the SPSR when in User mode or System mode is unpredictable.

MSR **Move to Status from Register**

Syntax
`MSR`⟨*cc*⟩ `CPSR_`⟨*fields*⟩`,` ⟨*arg*⟩
`MSR`⟨*cc*⟩ `SPSR_`⟨*fields*⟩`,` ⟨*arg*⟩

where ⟨*arg*⟩ can be either an Immediate value or a register (see section 5.1.1 on page 51). ⟨*fields*⟩ is one or more of the following field specifiers:

`c`	Control field	bits 0–7
`x`	Extension field	
`s`	Status field	
`f`	flag field	bits 24–31

Operation
⟨*cc*⟩⟨*CPSR*⟩: CPSR(⟨*fields*⟩) ← ⟨*arg*⟩
⟨*cc*⟩⟨*SPSR*⟩: SPSR(⟨*fields*⟩) ← ⟨*arg*⟩

Description The `MSR` (Move to Status from Register) instruction copies the value of the source register (R*s*) or immediate constant to the CPSR or the SPSR of the current mode.

Which parts of the status register are to be modified are specified via the ⟨*fields*⟩ field.

Usage This instruction is used to update the value of the condition code flags, interrupt enables, or the processor mode.

The value of a status register should normally be updated by moving the register to a general-purpose register (using the `MRS` instruction on the preceding page), modifying the relevant bits of the general-purpose register, and restoring the updated general-purpose register value back into the status (using the `MSR` instruction). For example, a good way to switch to Supervisor mode from another privileged mode is:

```
MRS   R0, CPSR        ; Read CPSR
BIC   R0, R0, #0x1F   ; Remove current mode (lower 5 bits)
ORR   R0, R0, #0x13   ; substitute Supervisor mode
MSR   CPSR_c, R0      ; Write modified register back
```

You should only write to those fields that they can potentially change. The `MSR` instruction in the above code can only change the control field, the other bits/field are unchanged since they were read from the CPSR by the first instruction. So it writes to `CPSR_c`, not `CPSR_fsxc` or some other combination of fields.

Although the state or extension fields, it is a good idea to write to these fields when writing the whole register. Rather than just writing to the flags and control fields (`CPSR_fc`) you shoud write to all four fields (`CPSR_fsxc`).

Note that due to a bug in the ARM Software Development Tookit, version 2.50 and earlier, you can only write to the flags and control fields.

The immediate form of this instruction can be used to set any of the fields of a status register, but you must take care to adhere to the read-modify-write technique. The immediate form of the instruction is equivalent to reading the register concerned, replacing all the bits in the relevant field by the corresponding bits of the immediate constant and writing the result back to the status register. The immediate form must therefore only be used when the intention is to modify *all* the bits in the specified fields. Failure to observe this rule might result in code which has unanticipated side-effects on future versions of the architecture.

As an exception to this rule, it is legitimate to use the immediate form of the instruction to modify the flags byte despite the fact that bits 24–26 of the status register have not been allocated at present. For example, this instruction can be used to set all four flags:

```
MSR  CPSR_f, #0xF0000000
```

Condition Codes

The current state will be effected by the value written into the current process status register (CPSR). If one of the saved process status registers (SPSR) is used the current status does not change.

Notes Any writes into the control, extension, and status fields (lower 24 bits) of the CPSR when in User mode will be ignored, so that User mode programs cannot change to a privileged mode.

As there is no saved process status register in User or System mode, any attempt to write into the SPSR while in User or System mode will be unpredictable.

MUL **Multiply**

Syntax `MUL`$\langle cc \rangle\langle S \rangle$ Rd, Rm, Rs

Operation

$\langle cc \rangle$: Rd $\leftarrow$ (Rm $\times$ Rs)(0:31)$\langle arg \rangle$
$\langle cc \rangle\langle S \rangle$: CPSR $\leftarrow$ ALU(flags)

Description The `MUL` (Multiply) instruction is used to multiply signed or unsigned variables to produce a 32-bit result. The condition code flags are optionally updated, based on the result.

Condition Codes

The N and Z flags are set according to the result of the multiplication. While the C flag should be left unchanged, a bug in the design means that the C flag is unpredictable after a `MULS` instruction.

Notes Specifying the same register for Rd and Rm has unpredictable results.

As the `MUL` instruction produces only the lower 32 bits of the 64-bit product, it gives the same answer for multiplication of both signed and unsigned numbers.

MVN	**Move Negative**

Syntax `MVN`⟨*cc*⟩⟨*S*⟩ R*d*, ⟨*op1*⟩

Operation

⟨*cc*⟩: R*d* ← $\overline{\langle op1 \rangle}$
⟨*cc*⟩⟨*S*⟩: CPSR ← ALU(Flags)

Description The `MVN` (Move Negative) instruction moves the logical one's complement of the value of ⟨*op1*⟩ to the destination register R*d*. The condition code flags are optionally updated, based on the result.

Usage `MVN` is used to:

- Write a negative value into a register.
- Form a bit mask.
- Take the one's complement of a value.

Condition Codes
The N and Z flags are set according to the result of the operation, and the C flag is set to the carry output bit generated by the shifter (see 5.1 on page 51). The V flag is unaffected.

Notes If the destination register (R*d*) is the program counter (PC or R*15*), then the current status register (CPSR) is restored from the saved status register (SPSR). This form of the instruction is unpredictable if executed in User mode or System mode, because these modes do not have an SPSR.

ORR	**Bitwise OR**

Syntax `ORR`⟨*cc*⟩⟨*S*⟩ R*d*, R*n*, ⟨*op1*⟩

Operation

⟨*cc*⟩: R*d* ← R*n* ∨ ⟨*op1*⟩
⟨*cc*⟩⟨*S*⟩: CPSR ← ALU(Flags)

Description The `ORR` (Logical OR) instruction performs a bitwise (inclusive) OR of the value of register R*n* with the value of ⟨*op1*⟩, and stores the result in the destination register R*d*. The condition code flags are optionally updated, based on the result.

Usage `ORR` can be used to set selected bits in a register. For each bit, `OR` with *1* sets the bit, and `OR` with *0* leaves it unchanged.

Condition Codes
The N and Z flags are set according to the result of the operation, and the C flag is set to the carry output bit generated by the shifter (see 5.1 on page 51). The V flag is unaffected.

Notes If the destination register (R*d*) is the program counter (PC or R*15*), then the current status register (CPSR) is restored from the saved status register (SPSR). This form of the instruction is unpredictable if executed in User mode or System mode, because these modes do not have an SPSR.

RSB	**Reverse Subtract**

Syntax `RSB`⟨*cc*⟩⟨*S*⟩ R*d*, R*n*, ⟨*op1*⟩

Operation

⟨*cc*⟩: R*d* ← ⟨*op1*⟩ − R*n*
⟨*cc*⟩⟨*S*⟩: CPSR ← ALU(Flags)

Description The RSB (Reverse Subtract) instruction subtracts the value of register R*n* from the value of ⟨*op1*⟩, and stores the result in the destination register R*d*. The condition code flags are optionally updated, based on the result.

Usage The following instruction stores the negation (two's complement) of R*x* in R*d*:

```
RSB     Rd, Rx, #0
```

Constant multiplication (of R*x*) by $2^n - 1$ (into R*d*) can be performed with:

```
RSB     Rd, Rx, Rx, LSL #n
```

Condition Codes
The N and Z flags are set according to the result of the subtraction, and the C and V flags are set according to whether the subtraction generated a borrow (unsigned underflow) and a signed overflow, respectively.

Notes The C flag is set if no borrow occurs, and clear if no borrow occurs. In other words, the C flag is used as a $\overline{\text{borrow}}$ (not borrow) flag. This inversion of the borrow condition is usually compensated for by subsequent instructions. The SBC (subtract with carry) and RSC (Reverse Subtract with Carry) instructions use the C flag as a $\overline{\text{borrow}}$ operand, performing a normal subtraction if C is set and subtracting one more than usual if C is clear. The HS (unsigned higher or same) and LO (unsigned lower) conditions are equivalent to CS (carry set) and CC (carry clear) respectively.

If the destination register (R*d*) is the program counter (PC or R*15*), then the current status register (CPSR) is restored from the saved status register (SPSR). This form of the instruction is unpredictable if executed in User mode or System mode, because these modes do not have an SPSR.

RSC — Reverse Subtract with Carry

Syntax RSC⟨*cc*⟩⟨*S*⟩ R*d*, R*n*, ⟨*op1*⟩

Operation

⟨*cc*⟩: R*d* ← ⟨*op1*⟩ − R*n* − $\overline{\text{C}}$
⟨*cc*⟩⟨*S*⟩: CPSR ← ALU(Flags)

Description The RSC (Reverse Subtract with Carry) instruction subtracts the value of register R*n* and the value of $\overline{\text{Carry flag}}$ from the value of ⟨*op1*⟩, and stores the result in the destination register R*d*. The condition code flags are optionally updated, based on the result.

Usage To negate the 64-bit value in R*0*, R*1*, use the following sequence (R*0* holds the least significant word) which stores the result in R*2*, R*3*:

```
RSBS   R2, R0, #0
RSC    R3, R1, #0
```

Condition Codes
The N and Z flags are set according to the result of the subtraction, and the C and V flags are set according to whether the subtraction generated a borrow (unsigned underflow) and a signed overflow, respectively.

Notes The C flag is set if no borrow occurs, and clear if no borrow occurs. In other words, the C flag is used as a $\overline{\text{borrow}}$ (not borrow) flag. This inversion of the borrow condition is usually compensated for by subsequent instructions. The SBC (subtract with carry) instructions use the C flag as a $\overline{\text{borrow}}$ operand, performing a normal subtraction if C is set and subtracting one more than usual if C is

clear. The HS (unsigned higher or same) and LO (unsigned lower) conditions are equivalent to CS (carry set) and CC (carry clear) respectively.

If the destination register (R*d*) is the program counter (PC or R*15*), then the current status register (CPSR) is restored from the saved status register (SPSR). This form of the instruction is unpredictable if executed in User mode or System mode, because these modes do not have an SPSR.

SBC **Subtract with Carry**

Syntax SBC⟨*cc*⟩⟨*S*⟩ R*d*, R*n*, ⟨*op1*⟩

Operation

⟨*cc*⟩: R*d* ← R*n* − ⟨*op1*⟩ − $\overline{\text{(C)}}$
⟨*cc*⟩⟨*S*⟩: CPSR ← ALU(Flags)

Description The SBC (Subtract with Carry) instruction is used to synthesise multi-word subtraction. SBC subtracts the value of ⟨*op1*⟩ and the value of $\overline{\text{Carry flag}}$ (Not Carry) from the value of register R*n*, and stores the result in the destination register R*d*. The condition code flags are optionally updated, based on the result.

Usage If register pairs R*0*, R*1* and R*2*, R*3* hold 64-bit values (R*0* and R*2* hold the least significant words), the following instructions leave the 64-bit difference in R*4*, R*5*:

```
SUBS  R4,R0,R2
SBC   R5,R1,R3
```

Condition Codes

The N and Z flags are set according to the result of the subtraction, and the C and V flags are set according to whether the subtraction generated a borrow (unsigned underflow) and a signed overflow, respectively.

Notes The C flag is set if a borrow occurs, and if clear no borrow occurs. In other words, the C flag is used as a $\overline{\text{borrow}}$ (not borrow) flag. This inversion of the borrow condition is usually compensated for by subsequent instructions. The RSC (reverse subtract with carry) instructions use the C flag as a $\overline{\text{borrow}}$ operand, performing a normal subtraction if C is set and subtracting one more than usual if C is clear. The HS (unsigned higher or same) and LO (unsigned lower) conditions are equivalent to CS (carry set) and CC (carry clear) respectively.

If the destination register (R*d*) is the program counter (PC or R*15*), then the current status register (CPSR) is restored from the saved status register (SPSR). This form of the instruction is unpredictable if executed in User mode or System mode, because these modes do not have an SPSR.

STM **Store Multiple**

Syntax STM⟨*cc*⟩⟨*mode*⟩ R*n*⟨*!*⟩, {⟨*registers*⟩}

where ⟨*mode*⟩ is one of:

IB	Increment Before	IA	Increment After
DB	Decrement Before	DA	Decrement After

Operation

```
if ⟨cc⟩
   IA: MAR ← Rn
   IB: MAR ← Rn + 4
   DA: MAR ← Rn − (#⟨registers⟩ × 4) + 4
   DB: MAR ← Rn − (#⟨registers⟩ × 4)

          for each register Ri in ⟨registers⟩
             IB:     MAR ← MAR + 4
             DB:     MAR ← MAR − 4
                 M(MAR) ← Ri
             IA:     MAR ← MAR + 4
             DA:     MAR ← MAR − 4

   ⟨!⟩:   Rn ← MAR
end if
```

Description The `STM` (Store Multiple) instruction stores a subset (or possibly all) of the general-purpose registers to sequential memory locations.

The register R*n* specifies the base register used to store the registers. Each register given in R*registers* is stored in turn, storing each register in the next memory address as directed by ⟨*mode*⟩.

If the base register writeback option (⟨*!*⟩) is specified, the base register (R*n*) is modified with the new base address.

⟨*registers*⟩ is a list of registers, separated by commas and specifies the set of registers to be stored. The registers are stored in sequence, the lowest-numbered register to the lowest memory address, through to the highest-numbered register to the highest memory address.

If R*15* (PC) is specified in ⟨*registers*⟩, the value stored is *unknown*.

Exceptions **Data Abort** — This exception is generated if the address (R*n*) is not word aligned, or is not accessible in the current processor mode.

Usage `STM` is useful as a block store instruction (combined with `LDM` it allows efficient block copy) and for stack operations. A single `STM` used in the sequence of a procedure can push the return address and general-purpose register values on to the stack, updating the stack pointer in the process. (See chapter 14 for further discussion.)

Condition Codes

The condition codes are not effected by this instruction.

Notes The base register R*n* should not be specified in ⟨*registers*⟩ when base register writeback (⟨*!*⟩) is used.

R*15* (PC) should not be used as the base register (R*n*).

STR **Store Register**

Syntax `STR`⟨*cc*⟩ R*d*, ⟨*op2*⟩

Operation ⟨*cc*⟩: M(⟨*op2*⟩) ← R*d*

Description The `STR` (Store Register) instruction stores a word from register R*d* to the memory address calculated by ⟨*op2*⟩.

Exceptions **Data Abort** — This exception is generated if the address (⟨*op2*⟩) is not word aligned, or is not accessible in the current processor mode.

Usage Combined with a suitable addressing mode, `STR` stores 32-bit data from a general-purpose register into memory. Using the PC as the base register allows PC-relative addressing, which facilitates position-independent code.

Condition Codes
The condition codes are not effected by this instruction.

Notes Using the PC as the source register (R*d*) will cause an *unknown* value to be written.

If ⟨*op2*⟩ specifies base register writeback (!), and the same register is specified for R*d* and R*n*, the results are *unpredictable*.

STRB **Store Register Byte**

Syntax `STR`⟨*cc*⟩`B` R*d*, ⟨*op2*⟩

Operation ⟨*cc*⟩: M(⟨*op2*⟩) ← R*d*(7:0)

Description The `STRB` (Store Register Byte) instruction stores a byte from the least significant byte of register R*d* to the memory address calculated by ⟨*op2*⟩.

Exceptions **Data Abort** — This exception is generated if the address (⟨*op2*⟩) is not accessible in the current processor mode.

Usage Combined with a suitable addressing mode, `STRB` writes the least significant byte of a general-purpose register to memory. Using the PC as the base register allows PC-relative addressing, which facilitates position-independent code.

Condition Codes
The condition codes are not effected by this instruction.

Notes Specifying the PC as the source register (R*d*) is *unpredictable*.

If ⟨*op2*⟩ specifies base register writeback (**!**), and the same register is specified for R*d* and R*n*, the results are *unpredictable*.

SUB **Subtract**

Syntax `SUB`⟨*cc*⟩⟨*S*⟩ R*d*, R*n*, ⟨*op1*⟩

Operation
⟨*cc*⟩: R*d* ← R*n* - ⟨*op1*⟩
⟨*cc*⟩⟨*S*⟩: CPSR ← ALU(Flags)

Description Subtracts the value of ⟨*op1*⟩ from the value of register R*n*, and stores the result in the destination register R*d*. The condition code flags are optionally updated, based on the result.

Usage `SUB` is used to subtract one value from another to produce a third. To decrement a register value (in R*x*) use:

```
SUBS  Rx, Rx, #1
```

`SUBS` is useful for decrementing a loop counter, as the branch instruction can test the (Z) flag for the appropriate termination condition, without the need for a compare instruction:

```
CMP  Rx, #0 BNE  Loop
```

This both decrements the loop counter in R*x* and branches back to the start of the loop (the `BEQ` instruction) if it has not reached zero.

Condition Codes

The N and Z flags are set according to the result of the subtraction, and the C and V flags are set according to whether the subtraction generated a borrow (unsigned underflow) and a signed overflow, respectively.

Notes

The C flag is set if no borrow occurs, and if clear a borrow does occur. In other words, the C flag is used as a borrow (not borrow) flag. This inversion of the borrow condition is usually compensated for by subsequent instructions. The `SBC` (Subtract with Carry) and `RSC` (Reverse Subtract with Carry) instructions use the C flag as a borrow operand, performing a normal subtraction if C is set and subtracting one more than usual if C is clear. The `HS` (unsigned higher or same) and `LO` (unsigned lower) conditions are equivalent to `CS` (carry set) and `CC` (carry clear) respectively.

If the destination register (R*d*) is the program counter (PC or R*15*), then the current status register (CPSR) is restored from the saved status register (SPSR). This form of the instruction is unpredictable if executed in User mode or System mode, because these modes do not have an SPSR.

SWI **Software Interrupt**

Syntax `SWI`⟨*cc*⟩ ⟨*value*⟩

Operation

⟨*cc*⟩: R*14_svc* ← PC + 8
⟨*cc*⟩: SPSR_svc ← CPSR
⟨*cc*⟩: CPSR(mode) ← Supervisor
⟨*cc*⟩: CPSR(I) ← 1 (Disable Interrupts)
⟨*cc*⟩: PC ← 0x00000008

Description Causes a SWI exception (see 3.4 on page 29).

Exceptions Software interrupt

Usage

The `SWI` instruction is used as an operating system service call. The method used to select which operating system service is required is specified by the operating system, and the `SWI` exception handler for the operating system determines and provides the requested service.

One of two methods are used: First, the ⟨*value*⟩ can specify which service is required, and any parameters needed by the selected service are passed in general-purpose registers. Alternatively, the ⟨*value*⟩ is ignored, and the register R*0* is used to select which service is required, any parameters are passed in other registers.

Condition Codes

The flags will be effected by the operation of the software interrupt. It is not possible to say how they will be effected. The status of the condition code flags is unknown after a software interrupt is *unknown*, unless specified by the operating system.

SWP	**Swap**

Syntax `SWP`⟨*cc*⟩ R*d*, R*m*, [R*n*]

Operation

⟨*cc*⟩: ALU(0) ← M(R*n*)
⟨*cc*⟩: M(R*n*) ← R*m*
⟨*cc*⟩: R*d* ← ALU(0)

Description `SWP` (swap) will swap a word between registers and memory. It loads a word from the memory address given by the value of register R*n*. The value of register R*m* is then stored in the memory address given by the value of R*n*, and the original value is loaded into register R*d*. If the same register is specified for R*d* and R*m*, this instruction swaps the value of the register and the value at the memory address.

Exceptions **Data Abort** — This exception is generated if the instruction attempts to access a part of memory which has been reserved for privileged mode access while the system is in user mode.

If a data abort occurs during the load phase, the store access does not occur.

Usage The `SWP` instruction can be used to implement semaphores. Where one process can send a message to another processes running on the same processor, or on a separate linked processor.

Condition Codes

The condition codes are not effected by this instruction.

Notes If the address contained in R*n* is non word-aligned the system will attempt an access, but the effect is *unpredictable*.

If the PC is specified as the destination (R*d*), address (R*n*) or the value (R*m*), the result is also *unpredictable*.

If the same register is specified as R*n* and R*m*, or R*n* and R*d*, the result is *unpredictable*.

SWPB	**Swap Byte**

Syntax `SWP`⟨*cc*⟩`B` R*d*, R*m*, [R*n*]

Operation

⟨*cc*⟩: MBR ← M(R*n*)(0:7)
⟨*cc*⟩: R*d*(0:7) ← MBR
⟨*cc*⟩: R*d*(8:31) ← 0
⟨*cc*⟩: M(R*n*) ← R*m*(7:0)

Description The `SWPB` (swap byte) instruction swaps a byte between registers and memory. It loads a byte from the memory address given by the value of register R*n*. The value of the least significant byte of register R*m* is stored to the memory address given by R*n*, the original value is zero-extended into a 32-bit word, and the word is written to register R*d*. If the same register is specified for R*d* and R*m*, this instruction swaps the value of the least significant byte of the register and the byte value at the memory address.

Exceptions **Data Abort** — This exception is generated if the instruction attempts to access a part of memory which has been reserved for privileged mode access while the system is in user mode.

If a data abort occurs during the load phase, the store access does not occur.

Usage The SWPB instruction can be used to implement semaphores, in a similar manner to that shown for the SWP instruction.

Condition Codes
The condition codes are not effected by this instruction.

Notes If the PC is specified for R*d*, R*n*, or R*m*, the result is *unpredictable*.

If the same register is specified as R*n* and R*m*, or R*n* and R*d*, the result also *unpredictable*.

TEQ **Test Equivalence**

Syntax TEQ⟨*cc*⟩ R*n*, ⟨*op1*⟩

Operation
⟨*cc*⟩: ALU ← R*n* ⊕ ⟨*op1*⟩
⟨*cc*⟩: CPSR ← ALU(flags)

Description The TEQ (Test Equivalence) instruction compares a register value (R*n*)with another arithmetic value (⟨*op1*⟩). The condition flags are updated, based on the result of logically exclusive-*OR*ing the two values, so that subsequent instructions can be conditionally executed.

Usage The TEQ instruction is used to test if two values are equal, without affecting the V flag, as CMP (compaire) does. The C flag is also unaffected in many cases. TEQ is also useful for testing whether two values have the same sign. After the comparison, the N flag is the logical Exclusive OR of the sign bits of the two operands.

Notes If the PC is specified for R*d*, R*n*, or R*m*, the result is *unpredictable*.

If the same register is specified as R*n* and R*m*, or R*n* and R*d*, the result also *unpredictable*.

TST **Test**

Syntax TST⟨*cc*⟩ R*n*, ⟨*op1*⟩

Operation
⟨*cc*⟩: ALU ← R*n* ∧ ⟨*op1*⟩
⟨*cc*⟩: CPSR ← ALU(flags)

Description The TST (Test) instruction compares a register value (R*n* with another arithmetic value (⟨*op1*⟩). The condition flags are updated, based on the result of logically *AND*ing the two values, so that subsequent instructions can be conditionally executed.

Usage TST is used to determine whether a particular subset of register bits includes at least one set bit. A very common use for TST is to test whether a single bit is set or clear.

Condition Codes
The N and Z flags are set according to the result of the operation, and the C flag is set to the carry output bit generated by the shifter (see 5.1 on page 51). The V flag is unaffected.

B ARM Instruction Summary

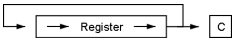
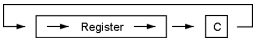
cc: Condition Codes

	Generic		*Unsigned*		*Signed*
CS	Carry Set	HI	Higer Than	GT	Greater Than
CC	Carry Clear	HS	Higer or Same	GE	Greater Than or Equal
EQ	Equal (Zero Set)	LO	Lower Than	LT	Less Than
NE	Not Equal (Zero Clear)	LS	Lower Than or Same	LE	Less Than or Equal
VS	Overflow Set			MI	Minus (Negative)
VC	Overflow Clear			PL	Plus (Positive)

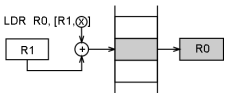
ARM Instructions

Add with Carry	ADC⟨*cc*⟩⟨*S*⟩	R*d*, R*n*, ⟨*op1*⟩	⟨*cc*⟩: R*d*	← R*n* + ⟨*op1*⟩ + CPSR(C)
Add	ADD⟨*cc*⟩⟨*S*⟩	R*d*, R*n*, ⟨*op1*⟩	⟨*cc*⟩: R*d*	← R*n* + ⟨*op1*⟩
Bitwise AND	AND⟨*cc*⟩⟨*S*⟩	R*d*, R*n*, ⟨*op1*⟩	⟨*cc*⟩: R*d*	← R*n* & ⟨*op1*⟩
Branch	B⟨*cc*⟩	⟨*offset*⟩	⟨*cc*⟩: PC	← PC + ⟨*offset*⟩
Branch and Link	BL⟨*cc*⟩	⟨*offset*⟩	⟨*cc*⟩: LR	← PC + 8
			⟨*cc*⟩: PC	← PC + ⟨*offset*⟩
Compare	CMP⟨*cc*⟩	R*n*, ⟨*op1*⟩	⟨*cc*⟩: CSPR	← (R*n* - ⟨*op1*⟩)
Exclusive OR	EOR⟨*cc*⟩⟨*S*⟩	R*d*, R*n*, ⟨*op1*⟩	⟨*cc*⟩: R*d*	← R*n* ⊕ ⟨*op1*⟩
Load Register	LDR⟨*cc*⟩	R*d*, ⟨*op2*⟩	⟨*cc*⟩: R*d*	← M(⟨*op2*⟩)
Load Register Byte	LDR⟨*cc*⟩B	R*d*, ⟨*op2*⟩	⟨*cc*⟩: R*d*(7:0)	← M(⟨*op2*⟩)
			⟨*cc*⟩: R*d*(31:8)	← 0
Move	MOV⟨*cc*⟩⟨*S*⟩	R*d*, ⟨*op1*⟩	⟨*cc*⟩: R*d*	← ⟨*op1*⟩
Move Negative	MVN⟨*cc*⟩⟨*S*⟩	R*d*, ⟨*op1*⟩	⟨*cc*⟩: R*d*	← $\overline{\langle op1 \rangle}$
Bitwise OR	ORR⟨*cc*⟩⟨*S*⟩	R*d*, R*n*, ⟨*op1*⟩	⟨*cc*⟩: R*d*	← R*n* | ⟨*op1*⟩
Subtract with Carry	SBC⟨*cc*⟩⟨*S*⟩	R*d*, R*n*, ⟨*op1*⟩	⟨*cc*⟩: R*d*	← R*n* - ⟨*op1*⟩ - $\overline{\text{CPSR(C)}}$
Store Register	STR⟨*cc*⟩	R*d*, ⟨*op2*⟩	⟨*cc*⟩: M(⟨*op2*⟩)	← R*d*
Store Register Byte	STR⟨*cc*⟩⟨*S*⟩	R*d*, ⟨*op2*⟩	⟨*cc*⟩: M(⟨*op2*⟩)	← R*d*(7:0)
Subtract	SUB⟨*cc*⟩⟨*S*⟩	R*d*, R*n*, ⟨*op1*⟩	⟨*cc*⟩: R*d*	← R*n* - ⟨*op1*⟩
Software Interrupt	SWI⟨*cc*⟩	⟨*value*⟩		
Swap	SWP⟨*cc*⟩	R*d*, R*m*, [R*n*]	⟨*cc*⟩: R*d*	← M(R*n*)
			⟨*cc*⟩: M(R*n*)	← R*m*
Swap Byte	SWP⟨*cc*⟩B	R*d*, R*m*, [R*n*]	⟨*cc*⟩: R*d*(7:0)	← M(R*n*)(7:0)
			⟨*cc*⟩: M(R*n*)(7:0)	← R*m*(7:0)

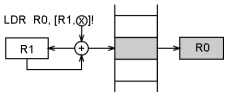
op1: Data Access

Immediate	`#`⟨*value*⟩	⟨*op1*⟩ ← IR(value)
Register	R*m*	⟨*op1*⟩ ← R*m*
		
Logical Shift Left Immediate	R*m*, `LSL` `#`⟨*value*⟩	⟨*op1*⟩ ← R*m* ≪ IR(value)
Logical Shift Left Register	R*m*, `LSL` R*s*	⟨*op1*⟩ ← R*m* ≪ R*s*(7:0)
		
Logical Shift Right Immediate	R*m*, `LSR` `#`⟨*value*⟩	⟨*op1*⟩ ← R*m* ≫ IR(value)
Logical Shift Right Register	R*m*, `LSR` R*s*	⟨*op1*⟩ ← R*m* ≫ R*s*(7:0)
		
Arithmetic Shift Right Immediate	R*m*, `ASR` `#`⟨*value*⟩	⟨*op1*⟩ ← R*m* ⇸≫ IR(value)
Arithmetic Shift Right Register	R*m*, `ASR` R*s*	⟨*op1*⟩ ← R*m* ⇸≫ R*s*(7:0)
		
Rotate Right Immediate	R*m*, `ROR` `#`⟨*value*⟩	⟨*op1*⟩ ← R*m* ⋙ ⟨*value*⟩
Rotate Right Register	R*m*, `ROR` R*s*	⟨*op1*⟩ ← R*m* ⋙ R*s*(4:0)
		
Rotate Right with Extend	R*m*, `RRX`	⟨*op1*⟩ ← C ⋙ R*m* ⋙ C

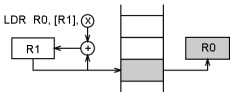
op2: Memory Access



Immediate Offset	[R*n*, `#`±⟨*value*⟩]	⟨*op2*⟩ ← R*n* + IR(value)
Register Offset	[R*n*, R*m*]	⟨*op2*⟩ ← R*n* + R*m*
Scaled Register Offset	[⟨*Rn*⟩, R*m*, ⟨*shift*⟩ `#`⟨*value*⟩]	⟨*op2*⟩ ← R*n* + (R*m* *shift* IR(value))



Immediate Pre-indexed	[R*n*, `#`±⟨*value*⟩]`!`	⟨*op2*⟩ ← R*n* + IR(value) R*n* ← ⟨*op2*⟩
Register Pre-indexed	[R*n*, R*m*]`!`	⟨*op2*⟩ ← R*n* + R*m* R*n* ← ⟨*op2*⟩
Scaled Register Pre-indexed	[R*n*, R*m*, ⟨*shift*⟩ `#`⟨*value*⟩]`!`	⟨*op2*⟩ ← R*n* + (R*m* *shift* IR(value)) R*n* ← ⟨*op2*⟩



Immediate Post-indexed	[R*n*], `#`±⟨*value*⟩	⟨*op2*⟩ ← R*n* R*n* ← R*n* + IR(value)
Register Post-indexed	[R*n*], R*m*	⟨*op2*⟩ ← R*n* R*n* ← R*n* + R*m*
Scaled Register Post-indexed	[R*n*], R*m*, ⟨*shift*⟩ `#`⟨*value*⟩	⟨*op2*⟩ ← R*n* R*n* ← R*n* + R*m* *shift* IR(value)

Where ⟨*shift*⟩ is one of: `LSL`, `LSR`, `ASR`, `ROR` or `RRX` and has the same effect as for ⟨*op1*⟩

Index